

Contents

GB Studio Comprehensive Guide	3
I. Basics & Getting Started	4
Basics: Backgrounds	4
Basics: Creating a HUD	6
Basics: Custom Events	10
Basics: Math Expressions	12
Basics: Math Functions	13
Basics: Switches	16
Basics: The Sprite Editor	18
Basics: Trigger/Positional Menus	22
Getting Started in hUGETracker	25
Getting Started with Games – Chapter 3: Cannon Defense	28
Version Control and GB Studio: Getting Started	33
Your First GB Studio Game	35
II. Development Workflow & Design Principles	36
Dealing With Burnout	37
Development Workflow - Chapter 1: Researching Your Game	39
Development Workflow - Chapter 5: Discipline & Demotivation	41
Development Workflow - Chapter 6: Readability (Part 1)	44
Development Workflow - Chapter 6: Readability (Part 2)	51
Development Workflow - Chapter 7: Color Theory	59
Development Workflow – Chapter 9: Creating a Color Palette (Part 1)	67
Development Workflow – Chapter 9: Creating a Color Palette (Part 2)	70
From Steps to Color Ramp	70
Dissecting Classic Cutscenes From the Game Boy	73
That was quite a Trip...	76
How to Make a Responsive Action Platformer	77
Making a Cutscene for the Game Boy in a Week	80
Solving the 'A Button Paradox'	83
Ten Lessons I learned During My Latest Project with GB Studio	85
Top Five Mistakes for First Time GB Studio Users	87
Zen and the Art of Retro Level Design in “Kudzu”	89
III. Music & Sound	89
FAQs for GB Studio Music	90
GB Studio Music Lesson 2: Noise Channel / Drums	91
GB Studio Music Lesson 3: Duty Channels	93
GB Studio Music Lesson 4: Wave Channel	94
GB Studio Music Lesson 5: Effects	96
GB Studio Music Lesson 7: Special Tricks Part 1	99
GB Studio Music Lesson 8: Common Mistakes	100
Getting Started in hUGETracker	102
VGM Sound Effects Part 3	106
VGM Sound Effects Part I.	107

hUGETracker Lesson 2 : Noise Channel	110
hUGETracker Lesson 5: Effects - Portamentos, Volume Control and Note Cuts	114
Effects Part 1: Portamento, Note Cuts and Volume Control.	114
Animating Background Tiles (3.0.3)	115
Basics: The Sprite Editor	117
Color Reproduction on GBC Compatible Devices and Emulators	121
Creating Multiple Dialog Frames using UI Palette Tricks	130
Development Workflow – Chapter 9: Creating a Color Palette (Part 1)	131
Development Workflow – Chapter 9: Creating a Color Palette (Part 2)	134
From Steps to Color Ramp	134
Lights Out - DMG Custom Color Palettes Tutorial	137
Managing Sprite Tiles	142
Sprite Style Differences in Game Boy and Other Consoles	155
Understanding Fonts	157
Advanced Trigger Techniques	162
Basics: Custom Events	165
Basics: Math Expressions	167
Basics: Math Functions	168
Dynamic Auto Scroller	170
How to Use QR codes to Enhance Your GB(C) Games	171
How to: Multi-Language Mechanic	174
Learning GBVM	177
Making an Isometric Game in GB Studio	183
Slopes in GB Studio	185
Unraveling The Thread Events	186
Common Thread	186
Variables and Maxing Them Out	190
VI. Hardware, ROMs & Emulation	192
A Guide to SGB Mode	193
Compiling GB Studio Roms into Standalone Executables	205
Converting your GB Studio ROMs for use with the N64	208
Distinguishing Reflashable Bootlegs	210
Game Boy Camera+ Mod	212
Getting GB Studio Games Running on the Xbox 360	214
Getting our GB Studio games running on the Nintendo DSi	215
How to Play a Game Boy Game on a Game Boy	216
Playing GB Studio ROMs in Minecraft	218
Porting GB Studio Games to Dreamcast	219
Possibly the WORST way to play Gameboy Games	221
Reflashing GBA Bootlegs with an NDS or NDS Lite	222
Running My GB Studio Games on Xbox	225
The Goomba Color Emulator	226
VII. RPG Series Tutorials	227
Let's Build an RPG: Quest 2 - Rewards & Level Up	228

The RPG Series	228
Let' s Build an RPG Quest 1-2: Enemy Turn, Effects, and Stats . .	231
The RPG Series	231
Let' s Build an RPG Quest 1-3: Animations & HUD	236
The RPG Series	236
Quest 3: Back to Town Part 1 - The Inn	240
The RPG Series	240
VIII. Game Genres: Puzzle	243
Let's Program a Puzzle Game! - Part 2	243
Setting Up The Electric Slide	243
Let's Program a Puzzle Game! - Part 3	249
Random Puzzle Generation	249
IX. Debugging & Testing	253
Testing Your Game on Different Hardware	253
X. Accessibility	257
Breaking Barriers: Develop GB Games For Gamers With Visual Im-	
pairments	257
Breaking Barriers: How To Develop Accessible GB Games For Gamers	
With Disabilities	259
XI. Publishing & Community	261
Basics: Switches	261
So You Want To Crowdfund? Part 1	263
Using Itch To Host Your Games	266
Version Control and GB Studio: Getting Started	268
Conclusion	269
Glossary	270
Quick Reference Cheat Sheet	271
GB Studio Editor Shortcuts	271
hUGETracker / GB Studio Tracker Shortcuts	271
General Tips	271
Index	271

GB Studio Comprehensive Guide

Welcome to the comprehensive guide for GB Studio, a user-friendly game development tool that empowers creators to build authentic retro games for the Nintendo Game Boy. Whether you' re a seasoned developer or a complete beginner, this guide is designed to navigate you through the fascinating world of GB Studio, from basic concepts to advanced techniques.

The Nintendo Game Boy, a nostalgic icon, is celebrated for its simple yet captivating games. GB Studio allows you to tap into this legacy, offering a visual scripting environment that abstracts away complex coding, making game creation accessible to everyone. This guide compiles a wealth of knowledge, tips, and tutorials to help you master GB Studio, covering everything from core

development workflows and sound design to advanced scripting and hardware deployment.

Prepare to unleash your creativity and bring your Game Boy game ideas to life!

I. Basics & Getting Started

- "Hello World" Four Ways
- Basics: Backgrounds
- Basics: Creating a HUD
- Basics: Custom Events
- Basics: Enemy Movement
- Basics: Flags
- Basics: Math Expressions
- Basics: Math Functions
- Basics: Recording with OBS
- Basics: Switches
- Basics: The Sprite Editor
- Basics: Trigger/Positional Menus
- Basics: Variables
- GB Studio Music Lesson 1: Terminology and Basics of the Game Boy Sound Chip
- GB Studio: Getting Started with Games – Chapter 1
- GB Studio: Getting Started with Games – Chapter 2 "Colorizing"
- Getting Started in hUGETracker
- Getting Started with Games – Chapter 3: Cannon Defense
- Version Control and GB Studio: Getting Started
- Your First GB Studio Game

Basics: Backgrounds

In the "Basics" series, we'll be going over all the technical foundations needed to start building a GB Studio game. If you're a GB Studio Beginner, then these articles are for you, and veteran devs can still get a refresher on the tools and concepts. # Understanding Backgrounds Backgrounds are a fundamental part of the GB Studio system. Understanding how to make functional backgrounds within their limitations is pivotal in successfully creating a game. In this article, we'll go over the basic concepts about backgrounds in GB Studio and how you can optimize and improve your game.

Let's imagine Scenes in the Game World are like theatre stages, and the backgrounds are the decorations. Of course, you need to make the decorations to make your theatre stage look amazing, but how do you go about that? Can you just create the background using a simple drawing program and forget about it? Unfortunately, it doesn't work like that in GB Studio. # How Backgrounds Work In 8-bit consoles like the Game Boy, resources were limited. When it comes

to backgrounds, this means you have a limited number of “tiles” you can have in every scene.

Let’s take a look at how they work: An example of a Gameboy game background scene.

This is a small quick background made in Aseprite. It’s a cat waving “hello” with the Chinese word for cat (mao) inside a border (and my signature in the bottom right). Now, let’s see how the Game Boy processes background into something it can display. An example of a background scene from GB studio. The image has a grid overlaid with an 8x8 grid.

The Game Boy uses *tiles*, which are small 8 by 8 pixel blocks. This is similar to how many 8-bit and 16-bit consoles stored graphics. Now, the Game Boy’s VRAM (Video RAM) is very limited, if we try to load this entire background onto VRAM with unique tiles (plus other things like sprites and text), there won’t be enough space and it will “overflow” or cause the wrong tiles to load. However, GB Studio already has tricks in place to prevent this.

Notice how most of the background is blank space? What if we store only one tile of blank space and then repeat that tile across the background? This is called *optimization*, and it’s very useful.

As previously mentioned, resources are limited, so developers had to optimize their games so they take up as few resources as possible. Let’s zoom in on the “hello” text bubble. A zoomed in image of a gameboy game background. The text says “Hello” .

As you can see, the border of the bubble and both “L” characters look the same. If all the repeating tiles are removed, this is the result: The image of the “hello” background, but now the repeating tiles have been removed.

Now, instead of 50 tiles of data, only 25 tiles need to be stored! That’s quite an improvement, isn’t it? Now, let’s turn our attention to the full image: A larger image of the previously shown background image.

When optimized for repeating tiles, this is the end result: The example background image, but with repeating tiles removed.

Pretty neat, huh? Luckily, you don’t have to manually go in and figure out all that by yourself, there are some free tools out there to help you out. # Using Apps to Count Tiles There are tools you can use to count tiles. One of them is Momeka’s Gameboy Sprite Exporter.

To load your background, click on the “Load Image” button and open your .png file. Then click “Remove Duplicates” . You should see your background with some red tiles. These red tiles are duplicates; optimized tiles, and the number of unique tiles is shown on the bottom left. An image of a background scene image being loaded into the Gameboy Sprite Exporter tool.

Another great tool for this purpose is Pautomas’ Tile Counter. It does the same

thing as The Sprite Exporter but in a convenient Web Interface. This is great if you don't have a computer that can open the Sprite Exporter, but this tool doesn't show you an overview of your graphics with the duplicates outlined in red.

As you can see, the example background only uses 98 tiles. That's quite a lot, but don't worry, backgrounds can use up to **192 tiles**. If you're not using dialogue, that number rises up to **256**. Is your number of tiles in your background still too high? Don't worry, here are some tips to help you: # Look for Similar Tiles An image showing an example of two background tiles that are very similar, but not exact duplicates.

Tiles that look very similar and don't really matter can be duplicated. Copy and paste one tile onto the other in a drawing app of your choice. # Remove Tiles with Few Pixels An example of a background area, where a tile has a single pixel. This tile placement could be optimized to save space.

Tiles that only have one or two pixels that are not important can be removed or painted with a single colour. This may seem problematic to your art when zoomed in, but when viewed on a Game Boy screen, it will unlikely to be noticed. Remember, the player won't know what you've removed, only what you're presenting. # Simplify Your Scene If your scene is busy or complicated, you should simplify it a little bit. Remove some less important elements and try out the previous tips.

I hope this helped you understand a bit more about the Game Boy and GB Studio. A lot of first time GB Studio users have challenges when first working with backgrounds because of their lack of understanding in how the Game Boy processes the backgrounds. Look for more guides in the future to help figure out sprites, text and scripting. SōdoDev

SōdoDev

a/k/a "Anima" – Game Developer Wannabe, Average Internet Enjoyer, Human Being. (he/him) ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### *Related*

Basics: Creating a HUD

To make the most out of a game, it's a good idea to have a clear representation of health, your amassed wealth of coins, or any number of things. You could create a menu to show this information, hiding it away from the main play screen

and keeping everything clean, but sometimes you want to leave that info on the screen, available at a glance. Providing a player with a visual indication of their progress or current status is an excellent way to communicate to your players, and it's pretty easy to pull off.

In this article, we'll cover the basics of getting a HUD set up for showing the player's health, discuss what makes it work and talk about some ways that code can affect its appearance, such as changing the sprite or hiding it. For this article, I am using the Sample Project for the base artwork. Two images showing the sample game from GB Studio. The image on the left is the standard game. The image on the right is the same, but now has a HUD.

Sample platformer scene as per the original template (left) and with the HUD setup (right). **# Pinning an Actor** To begin, our HUD is made by pinning actors to the screen. Move your actor to where you want it to stay on the screen, then click the pin in the top right of the actor settings. Finally, click the Pin to Screen checkbox to make it stay there. An image showing an actor is pinned in GB studio.

Pinning the HUD actor to the Screen. Now that your HUD actor is pinned in place, you need to determine what you want to do with it. Do you want it to represent your player's health, or should it be some other kind of indicator? The most common use for it is to show a health meter, so our example will help you set up a system like that. **# Setting up your sprite** To make a health bar or system that changes when you get hit, there are two ways to do it: 1. Swap sprites between different sheets, or 2. Make one sprite sheet that has several frames. My favorite way of doing this is by setting up a sprite that is sectioned out in several frames, so let's go ahead and set that up. Remember that our sprite is 16×16 pixels in height and width, but we're going to be adding additional width to the image file to create additional frames for it. An image showing a sprite sheet in Aseprite. There are 5 frames of animation for the HUD actor.

A single 16×16 pixel sprite made up of 5 separate frames. The art program I prefer is called Aseprite, but you can use whatever software you feel most comfortable with. You'll notice that I already have my sprite set up in this picture, and for each 'frame', you're going to want to add 16 pixels to your image's width. So in this example, my image file size is 80 (16 x 5 = 80) x 16 pixels. Remember to make the background transparent so it appears to be floating above the action.

If your program offers it, *turn on guidelines or a grid* to show you where the edge of each sprite is (and remember to set it to 16 pixels), as it will help to ensure you're 'drawing within the lines'. Draw your first frame and then edit the other ones to show the meter or amount going down in each frame. You don't have to use five frames either; get creative, and make sure it's clear what the HUD element means or your player might not understand it! **# Setting up the Health variable** Once you have saved your sprite into your sprites

folder inside of your project folder, you should now be able to assign it to your pinned actor. It's a good idea to set your animation speed to "None" now if you haven't yet, unless you plan on having your pinned actor animate while playing. For our case, you want to ensure it's set to none.

To be able to make our HUD move between frames and show the health decreasing, we have to put some code together to make it work. It's good practice to create a variable to represent the value of health that the player character has. We can use math functions to make it go up or down. Specifically [**Variable: Increment by 1**] to increase the variable's value or [**Variable: Decrement by 1**] to decrease it.

Let's go ahead and set up our Health variable. We're going to name it "Player Health" and have it set *On Init* in this scene. That means it will set to this amount every time we load this scene, so do keep that in mind! An image of the Set Variable to Value event in GB Studio.

Setting our health value variable. My example sprite has 5 frames (frames 0 through 4), so I am going to set the value to 4 and move through the frames each time that goes down, all the way to 0. To do that, we need to check when the player is hit, and then decrease the value of *Player Health*. # **Getting Hurt** GB Studio can handle collisions by enemies in two ways: in the Scene or on the enemy Actor. We're going to put it directly on our Actor, the evil Turnip. Once you pick the *Collision Group* for your enemy (we're using Group 1), then you can set up the code within the *Player* section of the *On Hit* tab. An image showing a script written in the 'On Hit' section of an actor in GB Studio.

Setting up the enemy's On Hit script Let's start the breakdown of the script by explaining the [**If Variable Is 'False'**] part. You need to add a check to see if you've been hit, otherwise, it will restart the script each time you are hit, registering multiple hits in a row, and will disregard any Wait timers. I used the *Local 0* variable and renamed it to 'Is Hit' to accomplish this. By checking to see if 'Is Hit' is False and then immediately setting the value to True, I force it to finish running my entire script before setting it to False again (restarting the script). You'll also notice I added a [**Camera: Shake**] to the script to visually indicate to the player that they've been hit.

Each time you're hit, it will decrease the value of the 'Player Health' variable by 1. Even though I set my [**Wait**] to 2 seconds, you can change the length to whatever you desire if it is too long or short for what you need.

A good piece of advice here is to make sure that each enemy you want to hit the player is updated with a script like this, otherwise only the first enemy Actor you changed will affect the HUD. Now that we have that set up, let's make it actually *do something*. # **Making damage affect the HUD sprite** To tie the damage we are receiving from the Turnip to our HUD, we need to set up our pinned HUD Actor's script. Again, we have a couple of options for how to do this. You could use either a Switch or check the value in *On Update* and swap frames when it finds a change in value. We'll stick with the less complicated of

the two and just check the variable value. An image of the script written in the On Update area of an actor.

Setting Frames with Variable Values. An image continuing the script written On Update.

Setting Frames with Variable Values (continued). We're going to use [**If Variable Compare With Value**] and the *Else* command to set up a chain of checks to determine what frame we should show. Please note that each check that follows is *nested inside* the previous one's *Else branch* – this is required so the *On Update* script isn't running multiple checks (which can cause slow down).

Since my initial health is full and the HUD Sprite's animation frame 0 shows full health, I don't necessarily have to set the frame for this value, but just in case I decide to add a health pack feature to add health back, it's a good idea to have it here for continuity's sake. If you did everything right, then you should see the health value change each time you're hit by the Turnip enemy we set up.

Once the health value reaches zero, you should have something in place like changing the scene to a Game Over screen. Or perhaps you could get even fancier with setting up a lives system (again, something best suited for switches) and send them to Game Over after they are all gone. You could also place a sound effect in each [**If Variable Compare With Value**] check before the frame change to signify to the player that they've been hit. # **Hiding the HUD or swapping sprites** Let's say you had a cutscene where you wanted everything to look clean while you are showing off your characters moving and delivering their dialogue. You could use [**Actor: Hide**] to achieve this, just make sure you use [**Actor: Show**] after the cutscene is over or you'll lose your HUD elements!

Another thing you may want to do is allow for upgrades to your health bar and you can do that by just swapping the sprite with [**Actor: Set Sprite Sheet**]. This would likely be something you'd want to check a variable for during *On Init* so it swaps before the scene loads. Don't be afraid to get creative with it, it's a very versatile tool to have in your arsenal!

Good luck with setting up your UI! I can't wait to see what you all come up with.

The project file used here can be found on our itch page. Pearacidic

Pearacidic

Writer, Game Designer, Engine Bender, Retro Enthusiast. Co-creator of the In The Dark series and Grimace's Birthday. (He/Him)

pearacidicgames.com/ ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky

- Click to share on X (Opens in new window) X
 - Click to share on Facebook (Opens in new window) Facebook
 - Click to share on Mastodon (Opens in new window) Mastodon
 - Click to share on LinkedIn (Opens in new window) LinkedIn
 - Click to share on Reddit (Opens in new window) Reddit ### Related
-

Basics: Custom Events

As you continue working on your game, you may notice that several scenes have a lot of scripts in common. Perhaps you run events setting a player's sprite sheet and coordinates at the start of a new area, attach the same actions to your buttons, or a series of events to give out rewards after a battle. When you run into these situations, you probably find yourself copying and pasting the same events over and over again. Sure, placing them in an event group makes the cloning between actors/triggers/scenes more manageable – but when you need to make a bug fix or optimize a sequence, you'll find yourself losing time tracking down each copied and pasted script.

Luckily, there's an event type to make repeating scripts much easier: Custom Events! # What Are Custom Events? Custom Events are groups of events that you can create and add to your actor, trigger, and scene scripts just like normal events. The trick is that unlike the other standard events in GB Studio, *you* define what a custom event does. Once you've created a custom event, you can re-use that script multiple times, and whenever you make a change to a custom event, it will update for each occurrence across your game. Custom Events make managing repeated sequences much easier, as you only need to look in one place to test, edit, or fix a bug.

If you're creating the scripts anyway, how is this any different from simply creating the scripts directly in the actor/trigger/scene? While a Custom Event's biggest advantage is its ease of editing, there are some other uses. Let's dive into our first example to show what can be done. # Building a Custom Event For our first example, we'll build a custom event to replace a feature that has been “removed” from later versions of GB Studio. With the release of the 2.0 Beta, Actors no longer have a Movement Type, which removed the frequently used “random walk” option. Since that change, creating a randomly walking NPC or Enemy now takes a Math Event, A Switch, and several Move Relatives in an Actor's “On Update” tab. Sure, it's not the longest script, but it's also unlikely to change for each NPC you're using it in. This makes it a perfect candidate for a custom event. Instead of Copying and Pasting those events into another actor (or even cloning the actor altogether), we can create a “Move Randomly” custom event that will be called from the search list whenever we need to make an NPC move randomly.

To start, navigate to your Custom Events section. In Beta 5, you can find Custom Events via the Navigator (on the left side). If you've hidden this, you

can go to **View > Show Navigator** to bring it back. If you're on earlier versions, Custom Events appear when you click the background of the Game World (similar to how you view a player actor's settings). Custom Events can be given a name, and also a description that can be used as a reminder of what the event does.

Once created, you'll notice that the custom event field looks just like any other script. The major difference is when you add your events, variables and actors do not reference any that have been placed in your game. Instead, custom events use placeholder values, labeled alphabetically from A to J.

A single Custom Event can have up to 10 variable and 10 actor placeholders. When an event is added to a custom event's script, you can select which placeholder to reference. These placeholders can be renamed, and once that Custom Event is added to a scene, event or actor, then the **actual** variable or actor you wish to use can be assigned to that placeholder.

To continue our example, let's create that Walk Randomly custom event. Create a new Custom Event, and name it "Walk Randomly". In its script, add a [**Variable: Math Functions**] event. Select Variable A, and then have it set to a random number between 1-4. Then, add a switch, and set it to check variable A. In each of its "When" sections, add a move relative in a different direction with collisions enabled for "Actor A". An image of the Custom Event being created in the GB Studio interface. An image of a script written for a custom event. It is a Switch event with multiple options. In your scene, create a few actors, and then in their "On Update" field, add an event and search for "Walk Randomly" – you should see your custom event listed. When you add a custom event, it has a list of each placeholder variable. For this event, you should see a Variable A placeholder and an Actor A placeholder. Assign the variable to Local 0, and the actor to "Self" (these should be the default anyway). Do this for each actor in your scene, and then test out your game. An image showing how Custom Events can be searched in GB Studio interface.

Custom Events show up by their names. An animated scene in GB Studio. 3 characters are moving randomly, using the custom event.

That's too fast! (unless they're dancing) Oh no! They're moving much too fast. We'll need to make a change. Go back to your custom event's script, and place a "Wait" event at the end of it (a few seconds should do). Now, when you test your game, the change should be reflected for all actors that were using that custom event. A screenshot from GB Studio, showing a custom event being edited to now include a Wait event.

Add a wait to the script... An animated scene from GB Studio. The 3 actors are now moving slower.

That's better. # Limitations Custom Events are an excellent way to optimize your game when you find yourself repeating scripts. The quality of life they add to bug fixing and editing scripts across scenes is unmatched, but they do have

their limitations. Due to their limit of referencing 10 variables and actors, larger scripts (for example, being used in rpg battle systems) may need to be split across multiple custom events. You also need to make sure you're properly re-assigning the placeholder variables and actors once you add your custom event to an actor/scene/trigger. While they do improve the QA process for scripting, they have their own areas that will need to be reviewed for mistakes when adding them to scripts. # Other Common Uses This article covered how to use a custom event to replace a feature that is commonly used across game types, but there is no way to know exactly what scripts are being repeated in your game and how converting to custom events could help you in your development. In my own experience, I've had success with custom events being used to handle:

Button Assignment: attaching scripts to customize button mappings (like weapons, jumps, start menus, etc) can be handled via a custom event that is run at the start of every scene.

Managing Money: when working with variables over 255, running the scripts to pass values back and forth between your ones/tens/hundreds columns can be quickly referenced in a custom event, reducing your script complexity

Changing a player sprite sheet: if you have a character customization or use 4-frame animations, custom events can be called at the start of the scene or on update to make sure you're always using the correct sprite sheet.

Controlling Random Encounters, Puzzle Rocks, or other common elements of a dungeon in an RPG.

Updating HUD variables and displaying actors in an RPG battle system. Dan Enders

Dan Enders

Friendly Neighborhood Game Designer, Podcaster, and (ex)Librarian (he/him)
GB Studio Games | Tabletop Games ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### Related

Basics: Math Expressions

We're entering the summer season, and as the sun comes out and the weather warms up, there's only one thing on everyone's mind: math! School may be wrapping up, but that doesn't mean you're done with classes yet.

There have been many changes to features and functions in GB Studio since our last entry in the Basics series, and today we' ll be covering some of the new functions introduced in GB Studio 3.0+. ## What' s changed for Math Functions? Earlier in the Basics series, we covered the many ways you can manipulate your variables with standard events like incrementing or decrementing variables, as well as the more robust [Math Functions] event. If you' re new to GB Studio, you may want to review that article first. If you' re already familiar with these events, you' re in luck, very little has changed for the core math functions in GB Studio 3.0+. The main change that has impacted math in newer versions has been the addition of 16 bit variables, which increases the minimum/maximum values of your variables. Where necessary, older articles in the Basics series will be updated to make note of which functions are no longer limited to the previous cap of 255. ## New for 3.0: Math Expression Events GB Studio 3.0 introduced the following new Math events: * [Evaluate Math Expression] * [If Math Expression] These two events can be powerful new tools to add to your scripting. By using Math Expressions, you can write a single formula to handle math that previously may have required multiple stacked events to run! ## Writing Math Expressions Both of these new events allow you to write a "math expression" to manipulate your variables. An expression is a combination of numbers or variables and at least one mathematical operation. For example, "2 + X" (or, in GBStudio terms, 2 + \$Variable 1), is an expression.

Operations are the functions we use to manipulate numbers such as addition, subtraction, multiplication, etc. In GB Studio, we can use the following operations:
| | | | ***

Basics: Math Functions

As discussed earlier in our Basics series, Variables are a core component of making a game in GB Studio. To use variables to their full potential, it' s important to know which events you can use to manipulate variables, and the limits to each event' s usage. There are 15 events in GB Studio that directly deal with a variable' s values, not including more advanced events that store elements in variables. Four of these events are connected to the concept of Variable Flags, and will be covered in a future article. In this article, we will review the standard variable events, comparing variables, and the multiple features found in the Variable: Math Functions event. # Standard Variable Events ## Variable: Increment by 1 / Variable: Decrement by 1 Incrementing a variable increases its value by 1 each time the event runs, to a maximum of 255. Decrementing a variable decreases its value by 1 each time the event runs, to a minimum of 0. Incrementing and Decrementing variables can be a quick way to manage variables that track items or simple counters, and the built in clamp can be quite convenient. However, Incrementing and Decrementing are limited by only being able to adjust a variable' s values by 1 at a time. ## Variable: Set to True / Variable: Set to False / Variable: Set to Value Variable values can also be set directly. An important consideration for working with values is that True

and False are not special settings for a Variable, they are simply binary values. When a variable is 0, it is false. When a Variable has a non-zero value, it is true. Setting a Variable to True changes its value to 1, while setting it to False changes its value to 0. Setting a Variable to a Value changes a variable from its current value to the new specific value. Setting variables to true and false are useful for tracking simple states (whether something has been interacted with, if an objective is complete, etc). Setting variable values directly is useful for managing variable states at the beginning of a game or scene, such as setting an HP value.

Math Functions for Variables All of the standard variable events can be replicated in a single event: [**Variable: Math Functions**]. This event requires more input to work, but also introduces more options to all the events listed previously. Using the first drop down field on the left lets you choose the function for the event. The second drop down on the right provides the options for this function (more on these options below). The 6 functions for a Math event are:

- * **Set To:** Sets or defines your variable
- * **Add:** Adds to the variable
- * **Subtract:** Subtracts from the variable
- * **Multiply:** Multiplies the variable
- * **Divide:** Divides the variable
- * **Modulus:** Runs a Modulus function on the variable (more on this below!)

With Variable Math, you can Add, Subtract, Multiply, and Divide the values of a Variable. These functions operate exactly as they do in normal mathematics, but there are a few GB Studio concepts to take note of when using them:

- * Unlike with Incrementing and Decrementing variables, when using Variable Math, a Variable's values will **wrap** if they exceed 255 or go below 0. If you want to limit your variable's values to the minimums and maximum values (0 and 255), you'll want to check the "Clamp Value between 0 and 255" check box in the event.
- * Division in GB Studio will always round down. One of the benefits of using math functions over standard variable events, is that you can specify the values you'd like to add/subtract/etc. This gives you more options than incrementing or decrementing by 1, but you are not just limited to working with Values. Using the second drop down box on the right, you can choose from 5 options when using a math function.
- * **True:** This option is the same as a value of 1.
- * **False:** This option is the same as a value of 0.
- * **Variable:** This allows you to select another variable for the math operation to be performed with.
- * **Value:** Allows you to enter a specific value to use.
- * **Random:** Allows you to enter a range of values, and GB Studio will use a random value from within that range. Note: The ranges are Inclusive. For example, using a Math Function event like the one below is the same as using an "Increment Variable by 1" event, because we're only adding 1 to the variable and we're clamping the result at 255. The value can be set to any number however, allowing for far more flexibility. You could instead use the "Random" option to add a random number: Note: When using the random options, be sure to review our article on how GB Studio handles randomness, as the behavior may appear different than what you expect. And when you start using multiple variables, you can combine your variables to do new things. Let's say you have a battle system and you want to increase your player's damage based on their strength stat. If you're tracking the player's stats separately, you can modify the damage in your battle system using math events:

While the “Set To” option changes a variable’s values directly just like the “Set Variable to Value” event, using it in a math function gives you more control over what you set a variable to. Just like with addition/subtraction, etc, you can set a variable to True/False, a set Value, a random number, or even another variable’s value! The final option you have for Variable Math is the **Modulus** function. A *modulo operation* divides one number by another, but it returns the **remainder** of the division. For example, performing a normal division operation of 95 by 10, the result is 9.5 (in GB Studio, this would display as 9 since the engine always rounds down). If you were to use a Modulus operation: 95 mod 10, it would display a result of 5 since there is a remainder of 5 left over from the division. For another example, 12 mod 10 would result in 2, and so would 12 mod 5. This may sound complicated at first, but it is a very useful function when using variables to store information. For example, how many coins a player has: 99 coins can be thought of as 9 tens and 9 ones, and a modulus function can check the ones value. Don’t worry if you don’t see an immediate application for Modulus in your game just yet, more advanced tutorials will make use of this function. # Variable Checks Once you’ve started working with and manipulating variables, you’ll likely want to use them to check if certain conditions have been met. By using the events to check your variables, you can control which scripts run in your actors, triggers, etc. The most straightforward ways to check variables are to see if they are True or False. This is done using the [If Variable is True] or [If Variable is False] event. Remember, all variables are false if their value is 0 (and all variables start at 0), and a variable is true as long as its value is a non-zero number. The second way to check your variables is to compare them. There are two compare events: Variable Compare to Value, and Variable Compare to Variable. Both events have a similar setup, the event compares the current value of the selected variable, and checks if it is: * **Equal to** a Value/Variable * **Less than** a Value/Variable * **Less than or Equal to** a Value/Variable * **Greater than** a Value/Variable * **Greater than or Equal to** a Value/Variable Comparing variables has many uses! You can check if a player has found a certain number of items, if an NPC has been talked to a certain number of times, if a player can afford to purchase an item from a shop, etc. Comparison events are the backbone of many RPG battle systems, too. If you find yourself checking the same variable with multiple comparison events, you may want to use a switch in its place. You can learn more about switches in *this article*. If you want to download the ROM and GB Studio Project file we created for the demonstration in this article, check out our Itch Page. Dan Enders

Dan Enders

Friendly Neighborhood Game Designer, Podcaster, and (ex)Librarian (he/him)
GB Studio Games | Tabletop Games ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook

- Click to share on Mastodon (Opens in new window) Mastodon
 - Click to share on LinkedIn (Opens in new window) LinkedIn
 - Click to share on Reddit (Opens in new window) Reddit ### Related
-

Basics: Switches

In the “Basics” series, we’ ll be going over all the technical foundations needed to start building a GB Studio game. If you’ re a GB Studio Beginner, then these articles are for you. Seasoned GB Studio devs may still find different ways to tackle challenges with these articles, so it should be worth the read. As you develop your game and work with variables, you may have encountered situations where you’ ll need to check a variable for specific values. In some cases, the [Compare Variable to Value] event works quite well – but when you need to check multiple values, you may find yourself stacking many Compare Variable events. Before long, your scripts start looking like this:

If, then, If, then, If, then... While there’ s nothing wrong with stacking multiple compare events, this can quickly get out of hand. The more comparisons you need to make for specific values, the longer your script will become, creating more room for error and making your scripts harder to review. Luckily, an event exists specifically for this purpose! # Switches A switch is a single event that checks the value of a variable, and then runs a script when the variable matches an specified value. You can think of a switch as a way to combine multiple [If Variable Equal to] events within a single event. Before getting into use cases, let’ s dissect a Switch.

- Every Switch is connected to a single Variable.
- A switch can check for **up to 16** specific values, which are set by its **Number of Options** sections.
 - These 16 values start at 1 and increase in ascending order as you add additional options, but you are able to change the values to any number within variable limits (0-255), and place them in any order.
 - A Switch also has an “Else” section, which you can use to check for values beyond these 16 options. Like other events, you can also disable the Else if you don’ t need it.
- After setting your Variable and the number of Options, you will see sections labeled “When: X” . These are your Compare Events. Set a **Value**, and when the Variable is **equal to** that value, the switch will run the script contained in that group.
 - Since switches are looking for an “Equal to” value, it will “break” and not check other values after it executes a script for a matched value. Using the “Else” section is useful for these situations. # Why Use a Switch? Aside from consolidating your script, Switches are excellent events to use for common situations where you’ ll be frequently checking a variable for *specific* values. Simplify Menu Options The

[**Text: Display Menu**] event can have a maximum of 8 options, and each option is set with a connected variable to a specific value between 1 – 8 (or 0, if using that option). Using a switch with the same variable as your menu allows you to control all 8 options with a single event! For example:

Rotating Dialogue If you’ d like your NPCs to rotate through dialogue options, you can use one of their **Local Variables** to count the number of times you’ ve spoken to them. Then, using a switch connected to that variable, you can create different dialogue options.

Whenever you interact with this NPC, it Increments its Local 0 variable by 1 (if you remember, variables start at 0 until set, so this will immediately set our variable to 1 the first time we talk to the NPC). Then, we use a Switch to check the same Local 0 variable. The first time we speak to the NPC, the variable will be 1, so it displays the first dialogue. The second time, the variable is increased again, so it will show the 2nd option. Each time after that, because we only have 2 options to check, it will run the dialogue in the “Else” section. You could expand this even more by adding more dialogue options, or by resetting the variable when you’ ve reached your else. Using Switches for dialogue can also be a great way to create branching dialogue. Creating switches based on the answers a player provides, or if other conditions are met, can allow you to create more engaging conversations with your characters. ## Scene Changes When changing scenes, you can use a Switch to create multiple options for where the player is sent. This can be as simple as using a switch to check if a player has found a key or completed an objective (much like a True/False check), or even randomizing the next room by using the Switch in combination with a [**Math: Set Variable**] to Random event with the range being your switch outcomes. ## Controlling States Much like controlling the dialogue an Actor displays, you can control their behavior using Switches. In our article on Enemy Movement, we cover an example on how to use a Switch in combination with other events to create an enemy movement pattern. You can also use switches to check for other states on a scene’ s Init. In this example, we can change our player’ s sprite between 2 different styles. We can track what “Look” our player selected using a Global Variable, and then use a switch to set the player’ s sprite sheet. To guarantee that the change persists between scenes, we can put the same switch on our Scene Inits, so this way when we change rooms, we still look the same. You can even have this happen on a button press – in an adventure game, you may want your player to cycle through different weapons, but keep the attack action assigned to the same button. In the example below, the player can rotate their Blaster’ s attack mode between a Close Attack and a Projectile Attack by pressing the Select Button. When the player presses the Attack button, (B), the script checks the variable for a mode using a switch, and performs the associated attack. # Wrapping Up Switches are powerful events that can be helpful for consolidating your scripts, but they’ re not a complete replacement for the Compare Variable event. Here are a few final tips for using switches: * Switches only check for “Equal To” values, you’ ll need to use Compare Variable events

for “If Greater than” or “If Less Than” comparisons. * Switches can only have 16 options and an Else – but you can still look for more values on the same variable by placing another switch in your Else section. * Switch values do not need to be 1 – 16. Your “When: X” values can be any value between 0 – 255. I do recommend keeping them in a certain order just to make it easier to review. * Combining switches within switches can be a great way to create varied Actor behavior! ## Practice Challenge: Using what you’ ve learned about Switches, create an Elevator that takes your player between multiple scenes. Dan Enders

Dan Enders

Friendly Neighborhood Game Designer, Podcaster, and (ex)Librarian (he/him)
GB Studio Games | Tabletop Games ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### *Related*

Basics: The Sprite Editor

GB Studio 3.0 introduced many new features. One of the most noticeable features is the new Sprite Editor. Those who are familiar with previous versions know that users used to be relegated to following strict sprite sheet guidelines in order for the engine to determine if the sprite was stationary, able to point different directions, or move around. These limitations were good for beginners, as they were forced into a narrow workflow and having a limited scope helps ensure that first time devs would be more likely to successfully complete projects. It became quickly apparent that these limitations also limited what kind of sprites can be made, and by extension what games were possible to create. Some clever workarounds and tricks were figured out by the community to include things like large sprites or 3 frame walk cycles, but all that is a thing of the past with the **Sprite Editor**. Users are able to create complex animations, large sprites, and many different states that help define how a game looks and feels. Many devs have already dove in and figured out the nuances of the editor, but if you’ re just starting out, or maybe you wanted to learn more about it from the ground up, it’ s time we properly introduced you to the features of the sprite editor and how to use it. ## Before Starting It’ s important to note that while the Sprite Editor is a great tool and can really unlock some awesome game development tools, it’ s not required to make a game. If you are a fan of GB Studio’ s default sprite behavior, you can keep using it without any detriment to your development. If this is your first game, it’ s probably best to stick to the default sprite settings too. For a refresher, read the official GB Studio Docs to learn about “Simple Sprites” and the four types you can create. You can also

read on in the documentation for how the Sprite Editor handles more complex sprites, otherwise continue reading our tutorial here. We’ ll be using the sample project, which is a great way to learn what’ s possible. ## Sprite Sheets The first thing to understand is how the sprite assets are determined by GB Studio. Users who are familiar with previous versions of GB Studio or other development tools might assume that the best way to manage different actions like running, jumping, and attacks would be created in different sprite sheets to keep things organized. In GBS 3 however, this is handled by *animation states* (more on that later). Ideally all the different poses, actions and variations of a sprite should be included on one large sheet. Think of each sheet like the pieces of a doll or action figure that you will be putting together. You can really break things down into small sections to build within the editor, or you can create whole sprites and animation frames. GB Studio will try to optimize the tiles used by each sprite, so duplicate or empty tiles and frames shouldn’ t take up more resources. Let’ s start our understanding of the Sprite Editor by looking at the Sample Project. You can create one in GB Studio by going to the **File Menu** and Selecting **“New Project”** (this dashboard also shows up when you launch GB Studio), then make sure the “Sample Project” is selected in the Template section.

GBS 3’ s template selection menu Now head to the View Menu at the top of the screen and select **“Sprites “**. On the top left side of the screen, you’ ll see a whole bunch of sprites listed in the *Sprite Navigator*. Each one of these corresponds to a PNG file that is stored in the project’ s **/assets/sprites** folder. If you want to create a new sprite, you will need to create a whole new sprite sheet PNG file or duplicate an existing one. You cannot simply add a reference to an existing sheet, which is contrary to how scenes and other assets work in the Game World view.

Every PNG file here corresponds to a sprite in the Sprite Editor. It’ s important to still work within the parameters of how GB Studio handles sprites, so make sure any sprite sheets you create use the correct colors, are at least 16 x 16 pixels in size and that the sprite sheet’ s height and width is divisible by a value of 8 to line up with the 8 x 8 tiles. Let’ s take a look at the **player_platform** sprite in the sprite list on the left. Clicking on it will bring up the *Animation Navigator* on the bottom left, the Canvas, Tile Selector and Frame Timeline in the middle, and a context sensitive details window on the right known as the *Editor Sidebar*. The panes in this window can be stretched and moved around to best suit what mode you are working in.

The Sprite Editor Window ## Animation States You can start navigating around and see how things work, but I find learning the tools are best understood by building something new, so let’ s try that now. We’ re going to change the way animation states are handled, so we can build some new states from scratch. Notice the *Animations Sidebar* on the left. It has only Animation states for the Right Direction. These specific states are the default for the Platformer game style, and we can see that is the *Animation Type* selected on the right sidebar. There’ s a checkbox under the Animation Type that says **“Flip ‘Right”**

To Create ‘Left’ Facing Frames” , uncheck that box and you will see the Animations Navigator suddenly includes three new states: **Idle Left**, **Moving Left** and **Jump Left**. The default behavior is to simply flip these frames in game to save resources, and that’ s perfectly fine for most games. There are some games, however, that require separate sprites for facing left and right, so by disabling that checkbox you can now create your own.

Samus’ weapon is on her right arm, so her left and right sprites are different. Select the newly created **“Idle Left”** Animation State from the navigator on the left and notice that both the *Canvas* and *Frames Timeline* are empty.

The “Idle Left” animation state has no frame or canvas data. ## The Sprite Canvas It’ s time to populate those areas. The Canvas is a preview of what the player will see in game, while the *Tiles Selector* shows what tiles are available to place in that Canvas. You can manipulate the size of the canvas on the top right along with the collision box, but this will remain consistent across all animation states. Move your mouse around the *Tile Selector*, and you will notice that a shadow appears on it, always two tiles high. This is important to understand. While the Game Boy works with 8×8 tiles, sprites themselves can be 8×8 or 8×16. For both simplicity and to save on resources, GB Studio only allows 8×16 sprites, so anytime you select from the Tiles area, it will have to be a multiple of that. Let’ s do this now. Click and Drag around the first animation frame to highlight it in a red box. Make sure you start on the top left of the first sprite image and drag to down to the bottom right. Once that area is selected in the Tiles, move your mouse on top of the *Canvas* and you will notice the selection appears allowing you to position it. Click on the canvas to set it.

Applying your selected tiles to the canvas field. Notice how the *Canvas* displays the sprite as four separate boxes? This is actually what’ s happening. You’ ve selected 4 8×16 sprite tiles from the *Tile Selector* and placed them adjacent to each other on the canvas. You’ ve just set your first frame! Now we have one problem. The sprite is supposed to be facing left. You could open an image editor, flip the frame, and append the sprite sheet with it, but GB Studio already has this built into the *Sprite Editor*. If you clicked off the canvas (like I did above) you’ ll want to re-select all four sprites in the canvas first. Notice if you try to click and drag like you did in the *Tile Selector*, it won’ t work as expected, you will instead move one of the 8×16 tiles in the canvas. Instead, hold Shift on your keyboard and click on the four different sprites to select them all. After you click the first one, you’ ll notice the *Sidebar* to the right changes, and new buttons appear. One of the buttons looks like a paper airplane pointing up, that button will mirror the sprites on the Y axis. Click that to flip the sprite.

Flipping sprites across the horizontal axis ## Frames Timeline You should now know how to pull frames into the canvas and manipulate them. There are many more buttons and features of the sidebar, but we’ ll have to save those for another time. For now, let’ s continue building this Idle Left animation using the *Frame Timeline* on the bottom. If we examine the Idle Right Animation State, we can see that the Idle Animation is actually 6 frames in length. If we

number the first three poses in the “**player__platform**” sprite sheet from left to right, then the order of the frames in the timeline are 1, 1, 2, 2, 1, 3 – the player stays still for 2 frames, drops their head slightly for 2 frames, returns to the first pose for 1 frame, then blinks their eyes for 1 frame. It is possible to speed up and slow down animation within the GB Studio game world with an event, but there is a limit to how slow things can animate, and if you have uneven animation (like above where the pattern is not evenly spread) then duplicating frames is necessary. Rather than doing all the work of adding another frame and placing the same sprite sheet pose as before, you can simply copy and paste the first frame in the *Frame Timeline*. Click on the first frame to highlight it. Then either go to the **Edit** menu and select **Copy** then **Paste**, or use the keyboard shortcuts Ctrl-C, Ctrl-V (Command-C, Command-V on Mac) to duplicate the frame. You now have a perfect copy of that first frame. For the next frame, you’ll want a blank canvas to start with, so press the “+” button in the *Frame Timeline*. Follow the steps as above, but this time select the second pose where the player’s head is slightly lower. Then go ahead and copy and paste that frame too.

Adding (and flipping) additional frames to the Frames Timeline You can also copy and paste frames out of order. So select the first frame (which is needed for the fifth frame now), copy and paste it – it will automatically go to the last frame position, which is where we need it. If you needed a pasted frame somewhere else, you can simply drag and drop it in the timeline. You can also delete unwanted frames as well using the “**Delete**” key on your keyboard. The last frame we need is the blink, so go ahead and create a new empty frame with the “+” button again and add the appropriate pose to the canvas.

Adding the final “blinking” animation frame for the Idle Left animation You can now preview your work using the play button in the top left of the canvas viewer.

Previewing the Idle Left animation in its entirety (on loop) We’ve only scratched the surface of what is possible with the Sprite Editor, but at this point you should have a basic understanding of what the workflow is like to create large and complex sprites in your game. 1. Create a Sprite Sheet and add it to the assets folder. 2. Go to the Sprite Editor and select the sprite. 3. Select the type of animation in the sidebar to set the default animation states. 4. Build your animations in the canvas using the tile selector. In the future, we’ll go over some of the more advanced features like creating custom animation states and calling them in the game, as well as creating multi color sprites and layering sprites. For now, go to town and have fun thinking about how you can use this powerful tool! Emi Paternostro

Emi Paternostro

Audio Engineer, Mac Technologist and Video Game Developer. Managing Editor of GBStudio Central. (he/him)

proximitysound.com ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
 - Click to share on X (Opens in new window) X
 - Click to share on Facebook (Opens in new window) Facebook
 - Click to share on Mastodon (Opens in new window) Mastodon
 - Click to share on LinkedIn (Opens in new window) LinkedIn
 - Click to share on Reddit (Opens in new window) Reddit ### Related
-

Basics: Trigger/Positional Menus

As you continue to develop your game in GB Studio, you will likely be creating menus for your player to check inventory, review a quest log, or edit settings. Although GB Studio has a built in **[Display: Menu]** event, you may find that it is too limiting for what you'd like to create. The *Display: Menu* event is limited to 8 options, and can only display 6 characters per item. Instead of trying to force this event to fit your needs, you can create scenes to function as your menu. In this tutorial, we'll cover how to create a Pause Menu within a scene that allows the player to navigate between 5 options. This will also go over how the Stack works for saving your player's position and returning to it. For this tutorial, we'll be using the "Top-Down 2D" scene Type, although it should also work for "Point and Click" style scenes as well. # Step 1: Creating a Menu Scene When making a trigger based menu, the majority of the menu information will be built into the scene's background itself. Creating your menu with a background has many benefits: you can create more than 8 options for the player, and you can change how you display this information. The example background in this tutorial may only have 5 choices, but the text displayed for each option is much longer than a **[Display: Menu]**'s character limits. With a background based menu, you can even choose to use icons or pictures instead of text for your choices. When creating your options, leave at least an 8x8 space between choices at minimum, since you don't want the triggers to overlap. # Step 2: Create a Cursor & Set Its Position In order to interact with the menu, you'll be treating the player like a cursor (cursor). Whenever your player enters the menu scene, change them into a cursor using the **[Set Player Sprite Sheet]** event. In this example, since the background text is aligned to an 8x8 grid, we've created a cursor sprite that aligns to the text. After changing the player's sprite On Init, use a **[Set Position]** event to place the cursor in the location where they'll begin in the menu. This position is where our first trigger will be, starting the navigation loop. # Step 3: Creating the Triggers A trigger based menu works by moving the player's position in between different triggers. Each trigger will check for the player's inputs to determine what to do next. This is accomplished using the **[Joypad Input: Pause Script Until Pressed]** event. First, create a trigger and place it where the cursor will be positioned. Then, in the Trigger's "On Trigger" menu, place a **[Joypad Input: Pause Script Until Pressed]** event. When this event fires, our game will pause until the selected buttons are pressed. If we combine this with **[If Joypad Input**

Pressed] events, we can control what happens based on the player's input. In this menu, our players can use **Up** and **Down** on the D-Pad to navigate, the **B button** to close the Menu, and the **A button** to interact with the menu option they've reached. With that in mind we'll need to create checks for those 4 inputs. After the Pause event, we'll create an event to control which scripts run based on which input was pressed:

To create the navigation, send the player to the position of the next menu choice using a **[Set Position]** event. By entering the coordinates of where the next trigger will be, you've now made it so that when the player presses **Down**, they'll move to the next trigger. Then, the trigger will engage, pausing the game again and waiting for the next input. If you copy+paste the trigger, and change the **[Set Position]** events for each new position, you can now move your cursor across multiple items: # Step 4: Interacting with the Menu Selecting options in the menu uses the same concepts as navigation. Within each trigger after the **[Joypad Input: Pause Script Until Pressed]** event, create an **[If Joypad Input Pressed]** event that checks for a press of the A button. In that event, you can then create a script to send players deeper into the menu (to check a quest list, for example) by using a **[Change Scene]** event. You can also use a scene change to send them back to the Title Screen for a "*Quit Game*" option. # Step 5: Attaching the Menu Now that we can navigate around the menu and use options, you'll need to give the player a way to enter and exit the menu on command. To create a "*Start*" menu, use the **[Attach Script to Button]** event to allow the player to press the Start Button and be sent to the menu scene. Before we change the scene to the Menu, we want to save the player's current scene and position so that they can return to that location after. This is accomplished by using the Scene Stack. By using the **[Scene: Store Current on Stack]** event before a scene change, we'll be able to return to this point later on.

Back in the Menu Scene, you'll want to give the player the option to back out of the menu. In each of our menu trigger's, when the B Button press is checked, add a **[Restore Previous Scene from Stack]** event. By using the scene stack, this will allow the menu to send the player back to whichever scene they entered from instead of returning to a specific scene like when using the Change Scene event. This will allow the player to return from the menu to the point where they paused in the game, regardless of which option they are currently hovering over. Now, with all these events and triggers set up, you'll have a trigger based pause menu that you can access by pressing Start during your game, navigate multiple options and pages, and then return to where you left off in the game from the Menu scene. If you're unfamiliar with the Scene Stack, keep reading. # Quick Concepts: Scene Stacks Scene Stacks, as mentioned above, are useful for transitioning your player between scenes that you may want to return from later. There are 4 events that directly interact with the scene stack. * Scene: Store Current on Stack * Scene: Restore Previous from Stack * Scene: Restore First from Stack * Scene: Empty Scene Stack To demonstrate how it works, let's look at our example menu system focusing on three scenes. Scene 1, Main

Menu and Quest Menu

Scene 1 The player begins in Scene 1. If they press Start, Scene 1 is loaded to the “Stack” using the [**Scene: Store Current on Stack**] event.

“Scene 1” is loaded to the stack. Then a Change Scene event is called and the Main Menu Scene is loaded:

Scene is changed to Main Menu

If the player selects “Quest List” from the menu, then “Main Menu” would be added to this stack using the [**Scene: Store Current on Stack**] before changing the scene to the Quest Menu:

Main Menu is now loaded to the stack.

Scene is changed to Quest Menu

With our example, if the player presses the “B” button from within this menu, then it will use the [**Scene: Restore Previous from Stack**] to return to the Main Menu, since it is in the top most location of the stack:

Main Menu is loaded since it’ s at the top. When this happens, it will not only change the scene to Main Menu, but remove “Main Menu” from the stack, bringing it back to the previous state.

The top of the stacked is “popped” off.

Now let’ s go back to the Quest Menu and add one more check for input. If a player presses Start, instead of Restoring Previous from Stack, it will call the [**Scene: Restore First from Stack**] event. This time, instead of restoring the scene from the top of the stack, it will load the one on the bottom, in this case the Scene 1 location:

Restoring First loads from the Bottom of the stack. Scene Stacking can be quite useful for creating menus that have sub menus or books with multiple pages. The Quest Menu in this example has multiple pages across separate scenes, but it will always remember the menu prior since it is saved in the stack. It’ s important to keep in mind that a scene stack doesn’ t automatically add scenes. If Scene 1 is stored to the stack and then the player moves to another Scene, using a restore event would take them back to Scene 1. To prevent this, use the [**Scene: Empty Scene Stack**] event on Init of an “in-game” scene. Dan Enders

Dan Enders

Friendly Neighborhood Game Designer, Podcaster, and (ex)Librarian (he/him)
GB Studio Games | Tabletop Games ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon

- [Click to share on LinkedIn \(Opens in new window\) LinkedIn](#)
 - [Click to share on Reddit \(Opens in new window\) Reddit ### Related](#)
-

Getting Started in hUGETracker

This series will cover the use of the hUGETracker software. hUGETracker is a standalone app native to the hUGEDriver which allows you to create music that is playable on the Gameboy. hUGEDriver is a powerful driver allowing you to compose dynamic Gameboy songs without eating up so many resources that it can't play music during a busy Gameboy game.

This article will briefly cover the installation and UI as well as taking a quick look at a sample song which explores what the Gameboy can do. This article is centered on using hUGETracker for GB Studio, but you can use it to create songs for GBDK, RGBDS and any other platform that can use .wav files. ## **Installation** hUGETracker is a cross-platform stand-alone tracker. This article will cover just the basic installation. If you are using a version of GB Studio that comes before version 3.2, you'll want to download hUGETracker version 1.0 beta 9.

This link will take you to the latest version to install for GB Studio version 3.2 or higher. Just unzip all the files into a directory of your liking, then run hUGETracker.exe. You can make a pinned or desktop shortcut if you'd like as well. ## **User Interface** If you've used other trackers such as OpenMPT or Famitracker, the hUGETracker interface will seem very familiar. We will assume you've never touched a tracker before, and start from scratch in this series.

Let's start with the basics. Download the [htt-demo.uge](#) here (the sample UGE was made in 1b9 version but its forwards-compatible with any other hUGETracker versions, once you save it in a later version though, you cannot go back), drop it somewhere on your hard drive and then go to File > Open... in hUGETracker and locate it. You should see something like the picture below. Don't be intimidated, we will soon learn what everything does and how it works. Before we go any further, let's learn what all the components are called. A screenshot of the hUGETracker interface, showing a song loaded into the tool. The main part of the screen (gray section) is the Tracker Grid. This is where you compose your song. Don't worry about what all the numbers and letters in the grid mean right now. Let's start by looking at the overall layout and how it works.

At the top you'll notice 4 columns labeled Duty 1, Duty 2, Wave, and Noise. These represent the 4 channels the DMG sound chip is capable of playing. We will look at them briefly here, but dive into more details later on.

Duty 1 / Duty 2 – These are simple Pulse channels. They make up the sounds you hear mostly as melodies and leads in Gameboy music. They can make a thin or fat simple 8-bit tone and have a few other features we will look at later on.

Wave – The Wave channel is what gives the Gameboy its distinct crunchy sound. More often than not the Wave channel provides the bassline or bass kick drum sound of a song. The wave channel is highly configurable and can make beefy noisy blasts or delicate smooth tones.

Noise – The Noise channel basically molds atonal static into sounds of drums, buzzing machines, explosions, bells, ocean waves sounds, and wind. You almost always use it for drums or special effects.

The lines below each channel represent a **pattern**. In hUGETracker, a pattern is a list of commands for a single channel. The four patterns will play simultaneously at the same time through the soundchip. We will get into the insides of a pattern in a moment.

Each pattern is made up of 64 rows. A **row** is a single moment in time in the song. It might start a note, stop a note, or make some modification to the song's properties. Each row's events across the four channels are executed simultaneously. The speed at which they occur is controlled by the tempo.

To the left of the Tracker Grid is the **Order Editor**. This is a high level view of your song. The highlighted gray background row is the row you're currently viewing in the Tracker Grid. If you click on another row, the Tracker Grid will display the patterns in that row. In the example on the screenshot, we are viewing the first order of the song, which contains patterns 0 1 2 and 3. The Order Editor's cells can be edited, you can change the number of a pattern to assign a different pattern to a certain channel for an order of the song. We will get more into this later.

Note: Although the UI sometimes refers to rows in the Order Editor as a "row," they are really called Orders. An Order is a set of 4 patterns that are played together.

Now let's dive into the internal components of a pattern. A screenshot from the hUGETracker user interface. The Note Cell, Instrument Cell, and Effect Cell are highlighted. Each of the four channels' patterns contain the same properties. The **Note Cell** tells the tracker what note to fire off when this row is reached. It might be a tone or a drum noise depending on the channel.

Next is the **Instrument Cell**, this tells the tracker what instrument to play the Note Cell's note through. We will learn about instruments later, but imagine them as a bass guitar, a drum, a trumpet, etc. The number corresponds to the instruments you see on the far left of the screen per channel.

Finally we have the **Effect Cell**. Effects do things like make a note quieter, change their stereo location, stop a note, or bend a note up or down. We will get way more into this later.

So to read things, let's practice, look at Row 20. When the tracker is playing a song and it reaches Row 20, it says "Ok, Duty 1, you play a F5 note out of instrument 1. Duty 2, you're gonna play a G#5 out of instrument 2. Wave

channel, I need you to fire off a A#4 with instrument 4' s wave. Noise channel, give me a D8 out of instrument 2." This all happens at once.

What does that sound like? Let' s learn how to play a song now and examine its pieces. ## **Playing A Song** At the top of the UI, you will see buttons that say Start, Cursor, and Stop. These control the playback of your song. Most composers will play chunks of the song, stop it, and make adjustments many many times (at least I do).

Pressing Start will start the song at the first pattern in the first row. Basically it plays the song from the beginning.

Pressing Cursor will start the song playing from whichever row is highlighted in the Tracker Grid. This may result in a song playing at a different tempo or not having all the notes playing until it has played for a few rows. It is handy when working on a long song and playing back just one section to review it.

Pressing stop will kill all sounds and put the cursor back where it was prior to pressing play or cursor.

Ok, now let' s hear some sounds. Press play and listen to some funky goodness.

As you can see, the highlighted line in the Tracker Grid shows you what' s playing and what you' re hearing as a result. As the song progresses notice the Order Editor also updates the Tracker Grid to show whichever order is currently playing. When it reaches the end it' ll start over. ## **Muting/Solo' ing Channels** Now let' s learn to isolate a channel. Isolating channels is useful when you' re not sure what' s causing an issue with your sound, or if you' d like to learn how another musician has compiled his channels to work together (it' s not always obvious). In music production speech, muting a channel means it won' t play along with the other parts of the song. Solo' ing a channel means the other channels will be silenced so you can only hear one.

Click the little sound speaker icon on all the channels except Wave. Your UI should now look like this: A screenshot of the hUGETracker interface, showing what happens when you isolate a channel. The red cross-out icon means the channel will not play when you press Play or Cursor. The data is still there, this is just for you to hear as you playback. Press Play now. You should hear the bassline with no other instruments.

Play around and solo each channel to understand how the song is composed. Hopefully you come to this conclusion:

Duty 1 – This plays the main lead. It has a kind of thick sound and is fairly loud. At the very end the pattern it plays some 8-bit tom tom drum notes in a little roll-styled bit.

Duty 2 – Duty 2 has a thinner sound than Duty 1 and is playing a harmonic version of the same lead. After a few orders, the harmony is dropped in favor of a quieter backing pad note which plays in a sustained way behind the lead of Duty 1.

Wave – The Wave is just playing the funky bassline. It is crunchy and curiously plays pattern 2 across all the orders.

Noise – The Noise channel uses a couple different instruments to produce a disco-styled beat. The beat pattern 3 also plays across all four orders. ##

When to use hUGETracker or GB Studio Tracker There are benefits to using GB Studio Tracker or hUGETracker depending on your needs. For short simple songs, and especially when you’re just starting out, GB Studio tracker is a lot less confusing to use. In GB Studio Tracker the concept of a pattern is a bit different than in hUGETracker but we won’t go into it here.

hUGETracker is better for songs that will be long and for which you desire to reuse some patterns along with other unique patterns within an order. It also has more export features you need if you aren’t using GB Studio (export to .gbs, .wav, .c, .asm).

Outside of exporting to other formats, the honest truth is there is nothing you can do in hUGETracker that you cannot do in GB Studio Tracker. You just have a little less granular control over patterns and orders in GB Studio Tracker, but you should use whichever you prefer. ## **Next Steps** So, although we’ve yet to put down a single note of our own design, we should understand fairly well how all the individual pieces come together. The next lesson will get into making your own song and first instruments as well as using the noise channel to make some simple drums.

This article is part of a series: hUGETracker Beatscribe

Beatscribe

Beatscribe has composed and produced soundtracks for games on Nintendo DS, Nintendo Switch, PS4, Xbox1, Sega Genesis and numerous iOs and Android games for over a decade, but his true passion is creating epic moods on ancient hardware like the Game Boy and NES. ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### *Related*

Getting Started with Games – Chapter 3: Cannon Defense

In this chapter we are using GB Studio version 4.1.3 to make a simplified version of Missile Command: a cannon at the bottom of the screen will fire projectiles at objects falling from the sky, with the goal of protecting the planet’s surface. We’ll achieve this using the “Launch Projectile” and “Attach Timer Script” commands.

You can find the project's source code at the end of this article, where you'll also be able to reuse the PNG tiles I've created or review the entire project if you have any questions. ## "The Shot" Let's begin with the most fun part: the shooting. The idea is that the cannon fires towards where we are pointing with the cursor. Normally, we would need to calculate the length of the two sides of the triangle formed and then find the arctangent, but since we are clever and lazy developers, thanks to Maltby, we can simply use the new "Launch At" function in the Launch Projectile command. We'll click on "Target actor" and set the Player as the target.

If you've never used the Launch Projectile command before, can you guess what it does? That's right, it shoots projectiles. It's the most commonly used command when we want our character to shoot projectiles. You need to select the sprite for your shot, who is shooting it, in which direction it's going, which collision group it belongs to, and which group it can hit. Here's how we would proceed: * Create a new scene. * Set the scene type to "Adventure" to take advantage of zone entry triggers (more on that later). * Add an actor to serve as the cannon at the bottom of the screen, for example, at coordinates x=9 and y=16. * Go to the scene's "On Init" section and add the command "Set Actor Collision disable" ; we don't want the Player (cursor) to interfere with our shots. * Add the command "Attach Script to A Button" and inside, place the command "Launch Projectile" and set the properties for the projectile as desired (I've set them as shown in the picture).

The important part is to select "Launch At" under the "Source" section of Launch Projectile and choose "Player" as the target * Finally, add a "wait" command of 0.5 after the "Launch Projectile" command to avoid overwhelming the screen with too many projectiles if using DMG mode, plus it adds a touch of difficulty. Now, if we try running the project, we should have a functional cannon that shoots towards the cursor. It would be a good idea to add a collision line from position (0,17) to (19,17) to prevent the cursor from reaching these areas, which would make it unrealistic to shoot downward. ## "The Meteorite" Now we arrive at the second fun part: the object to shoot at. We'll make an actor appear at a specific position and move in a straight line toward our planet. We'll place our new actor, for example, at position x=4 and y=1 (but later we'll move the sprite higher in the Sprite Editor so it appears from the top of the screen rather than suddenly in the middle).

You can adjust the collision zone as needed. In the actor's properties, we'll assign a collision group "1" and get to work creating the commands that will determine what happens when our shot hits it. The idea is that it disappears with a visual effect (we'll use another projectile for that). To do this, go to the "On Hit" tab for Group 3 (or the one you assigned to your projectile) and add the following commands: * "Variable set to true," selecting a new variable called Meteor1_destroyed. (This will be used in the next step to know when the actor can be replaced). * "Deactivate" and "Self," so it disappears from the scene. * (Optional) Launch Projectile with the meteorite as the source, a very

short lifetime, and directed upwards. The idea is for the object to disappear and show a visual effect representing its destruction. I used these parameters (see photo). * Finally, “Play Sound Effect,” which will play when the meteorite is destroyed. Everyone loves sound effects! Now we’re done with the script inside the actor, but there’s still one important part left: the movement and respawn of the meteorite. To do this, go back to the main scene script in “On Init” and use two commands that shouldn’t scare us: “Attach Timer Script” and the “Random” function. The Timer Script is a valuable resource for repeating a process multiple times. You specify how often it repeats, and inside, you can add all the processes that will occur each time it triggers. Here’s how we could implement it: * Add the command “Attach Timer Script.” Set the interval to 0.5 seconds (this is fine since it will only act if the meteorite is destroyed). Inside “On Tick,” write everything that should happen. * Start with the “If variable is True” command and select the previously mentioned variable, Meteor1_destroyed. * Inside the condition, the first thing is to use the command “Set variable to False” for Meteor1_destroyed, to revive it. Now comes one of my favorite resources: “Random.” We can give a variable a random value within the range we need. How exciting! What value will it get? * Next, add the “Math Functions” command, create a new variable (meteor_randomx), and set its “Value” to “Random” between 0 and 18 so it can appear at any position on the horizontal axis.

**If you need more help on how to use Random values or just want to see another example, check out this article: GB Studio: Getting Started with Games – Chapter 1 – GB Studio Central* * After that, use the command “Set Actor Position,” set the actor to Meteor, and click the “#” symbol to select a variable instead of a fixed value. The variable, of course, will be meteor_randomx. * Finally, use the command “Activate Actor” and activate the meteorite, which we’ve likely deactivated after firing. Then, with the “Actor move relative” command, move the Meteor actor 18 positions along the Y-axis (0,18). This command will move the specified number of spaces on the X and/or Y axis from its current position, regardless of where it is.

Our Timer Script should look something like this Now we’ll have a meteorite that falls toward our planet each time it’s destroyed. But what happens if we don’t manage to shoot it down before it reaches the surface? That’s what we’ll create next. ## “You Missed” You failed, and the meteorite has reached the surface. How about a good shake to get us back on track? We’ll check if the meteorite has made it to the end of its path: * After the “Actor Move Relative” command, use the “If variable is False” command, checking for Meteor1_destroyed. If it’s false, it means we didn’t destroy it, so inside the condition, we’ll include everything that happens: * Use the “Camera Shake” command and, if you like, maybe add a sound effect too. (The Camera Shake command is another favorite of mine; it creates a shake of customizable intensity). * “Deactivate Actor” to deactivate the meteorite so more can appear with our Timer Script, and set the variable “Meteor1_destroyed” to “True.” Finally, just add the “Variable set to True” command for Meteor1_destroyed at the start

of our program (just below where we disabled player collisions) along with a “Deactivate Actor” command for the meteorite, so that right at the start of the game, it will appear unexpectedly thanks to our Timer Script. With this, we now have a fully functional game! But if you want to embellish it with details like points, lives, and informative text, keep reading. ## Turning it into a Game The greatness is in the small details. In this section, we’ ll add some indicators or decorative elements. We’ ll introduce new variables to store the score and remaining lives. It will be easier to understand if we organize everything into Event Groups: * First, add an “Event Group” at the start of your main scene’ s “On Init” script and call it something like “init positions,” , it should look like this: * Next, create another Event Group called “init values,” where we’ ll reset all our variables. * Create a “score” variable and set it to 0. Create two more variables that will be used to split the score into two pairs of digits. In my case: “score(ten,uni)” and “score(tho,hun)” . We will explain how to use them later. * Create a “lives” variable and set it to 2. * As we did with the score, we will create a new actor to represent the lives and we will also set the animation speed to None. And then, we will assign the sprite to the variable \$lives. The thing is to open the sprite editor and arrange the different frames of the sprite in order: for example in frame 0, the sprite will be empty. In frame 1, one life icon will appear and in frame 2, two life icons will appear. * Finally, create a “restart” variable and set it to “False.” This will allow the “start” button to restart the game only when we actually run out of lives.

A wise piece of advice is to try to stay organized. It will make finding errors or understanding the project easier if you come back to it years later. * We will add a new actor which will serve as a visual aid to know when we can press the START button to start the game or even to warn when our game is over. We will simply create a sprite with a text that shows “Press Start” in frame 0 and Game over in frame 1. * After the group event, we will add a pause in the script using the “Pause Script until button pressed” command, so the game doesn’ t start until we press, for example, the “start” button. * After this command, we will add a small group of commands that will take place if the “start” button is pressed. We will achieve this using the “If Button Held” command, where the “start” button will be used. Inside, we will write the following:

1. “Seed random number generator” : It’ s a good idea to ensure the random variable works correctly.
2. Play Sound Effect: Add a sound effect that indicates the start of the game.
3. Play Music Track: Add an exciting background music track to create some excitement for the game. In my case, I used both SFX and music from free assets provided by our friend Beatscribe (visit his Itch.io profile where you’ ll find high-quality music and sounds):

* Finally, we will remove the “Press Start” actor, simply by using the command “deactivate actor” . ## High Score: If we want to keep track of points for destroying each meteor, we’ ll go back to the “On Hit” script, group 3 within our Meteor Actor, and add the following: * “Increment variable” : Add 1 to the “score” variable to increase the score. * Here we will perform the calculation required to separate a number that can reach up to 4 digits in two pairs of 2 digits. On the one hand we would

have the variable \$score(ten,uni) which would be equal to score%100 and the variable \$score(tho,hun) which would be equal to \$score/100. * We will use the Set Animation Frame command again but this time to represent numbers. Therefore we will have to set each frame of a sprite to show numbers from 00 to 99 (it is laborious but then you can reuse it for other projects). * So the actor responsible for displaying thousands and hundreds will be placed in the frame dependent on the variable \$score(tho,hun) and the one for tens and units on the variable \$score(ten,uni). * Finally, optionally, I wanted to add another “Switch” as a “level up” mechanic, increasing the meteor’ s movement speed according to the number of points. In my case, I used the “Actor set Movement Speed” command and gradually increased it by 0.125 at scores of 5, 10, 20, 30, 50, etc.

Will someone be able to reach 1000 points? ## Continue or Restart the Game: Almost there! We just need to add a few commands within our “Timer Script,” right after the commands included in the condition “If meteor1_destroyed = False” . The idea is that when the meteor is not destroyed and reaches the planet’ s surface, it should subtract lives and check if we still have any lives left to continue the game. * First, we will use the “Variable Decrement” command to reduce the “lives” variable by 1. * Next, we will compare the “lives” variable to see if it’ s less than 0 (if (\$lives < 0)), because if that’ s the case, the game will be over. Inside this condition, we will write the following: * First, we will add the “Stop Music” command to stop the music and cut the mood a bit, followed by a “Play Sound Effect” with a discouraging sound effect for our embarrassment. * We will change the value of the “restart” variable to True. This way, our start button (which we will modify below) will behave differently when “restart” is True. * We will also reset the meteor’ s movement speed to 1 (in case we’ ve surpassed a level during the game). This will be “Actor (meteor) Set Movement Speed” to 1. * Now we will assign the value 0 to the informational text sprite to show “game over” instead of “press start” . In the “Else” section within the condition “If (\$lives < 0),” we will handle the situation where we failed to stop the meteor but still have lives remaining. If there are still lives left, instead of “game over” we will perform the same process as before but this time it will be frame 0 which shows “Press Start” . * Immediately after, we will activate the text actor that will show “Game over” or “Press Start” depending on the case. * We will update the actor’ s frame representing the lives using the Set Animation frame command and make it depend on the \$lives variable. * Now, we will place the famous “Pause script until button pressed” and choose “Start” as the button to press in order to continue. After that we will deactivate the actor Text. Next comes an important step: we are going to add a specific script for the situation where we run out of lives (so, we have changed the “restart” variable to True). It would go like this: * We will add the command “If variable is True,” setting the variable in question to “Restart,” and inside, we will add the necessary commands to reset the game:

1. We will set the “restart” variable back to false. (variable set to false)
2. We will reset the score to zero. (Variable \$score set to Value 0)
3. We will have 2 lives again. (Variable \$lives set to Value 2)
4. And we will use the commands

again to update the Lives actor frame and the two score actors after recalculating the two digits again. That’s it, everything is done! Taking advantage of the “adventure” scene mode, we can add a series of trigger zones (by clicking the “Add Trigger” icon) with which we will change the cannon’s sprite so that it appears to move according to where we aim. This is a widely used technique that is achieved by modifying the animation speed of the Cannon Actor to 0 and using the “Set Actor Animation Frame” command with the different frame values within the Sprite editor for each trigger zone. **If you need help with managing sprites, take a look at this other article: GB Studio: Getting Started with Games – Chapter 1 – GB Studio Central* I hope you’ve managed to get this far successfully. Remember that you can download the project’s source code from <https://rubenretro.itch.io/cannon-defense>, and don’t forget that on the GBStudio Discord channel, there is a wonderful community that will be happy to help you with any questions.

Best regards to everyone, and see you soon!

This article is part of a series: Getting Started with Games RubenRetro

RubenRetro

Nurse. Husband. Hobbyist Game Developer using GBStudio. Cat lover.

rubenretro.taplink.ws/ ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### Related

Version Control and GB Studio: Getting Started

Have you ever used undo enough times to hit the program limit? Maybe you are saving multiple copies of your project as a way to create backups? Without a doubt, making games can get messy. There’s nothing worse than breaking a project, and having no way to go back to a working version of it. Thankfully, every GB Studio developer can use a tool to help them manage things: **Version Control**. Version control lets you go back in time to a previous version of your project. If some of your files go missing, your code breaks, or your files get corrupted, you can restore it to a previously functioning state. It’s a lot like using save states in a Game Boy emulator. If you’re afraid of losing a lot of progress by dying, you can make a save state to ensure you can go back and try again. In Git, a popular version control system, these save states are called **commits**. There are many services that utilize Git’s system for tracking changes, but for this guide we’ll be using GitHub, one of the most popular websites for version control. This guide will show you how to backup your

project, and make a **commit** when you want to update your project's state to the site. You will need a Github account (which is free for most of its features) and a GB Studio project. If you'd like to follow along with something risk-free, try using one of the Sample Projects. To create a repository for your GB Studio project, in Github: 1. Click the + at the top right of the screen 2. Click "New repository" 3. Name your new repository 4. Choose Public or Private 5. Click "Create repository" **Note:** *If your project has personal information, or assets you don't want to make public, make sure to set your repository to private.* You should see a new page for your empty repository. Github offers a lot of ways to upload your files, but right now we'll simply use the website. 6. Click "uploading an existing file" in the Quick setup box 7. Drag your entire project folder into the space provided 8. Type in a message for your commit 9. Click "commit" Commit messages are notes that help describe what you added to the project. This will make it easier to tell what's changed if you have to go back and find a previous snapshot of your repository. "Adds player shooting script" or "adds town background" are some examples. Look here for how to write *really* good commit messages. You have now backed up your GB Studio project to Github. If you want to update this project, click "Upload files" on your repository page, then repeat steps 7-9. **Note:** *I like to give my project files absurdly short names like "ttm_a1", yours should show up as whatever you named your project in GB Studio.* Using the Github website to update your files is not a great long-term strategy. Version control is best when you can make it a part of your project workflow. An example workflow could be: 1. Work on your GB Studio project 2. Open a program like Github Desktop 3. Make a commit 4. Click "push" This backs up your whole project privately to an external location, making it safe in case of hardware failure or broken code. Congratulations! You now know how to use Github to backup your projects and create commits to demark versions. Next time, I'll show you how to restore previous contents of your repository, and introduce you to using clients for git. This will make it much easier to rollback, move, and rename your project files if you need to.

Ember

Musician and GB Studio Composer. Manager of the GB Studio Community Assets Collection. Contributor to the GB Studio Documentation project. (she/her & they/them)

deertears.itch.io/ ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### Related

Your First GB Studio Game

When trying to tackle your first GB Studio project, there are some best practices that will greatly increase the chance of successfully completing your game. Many people start and end up frustrated or overwhelmed at the possibilities of what to make, or trying to figure out how to make their game work. We've compiled some advice from the GB Studio Central team and laid them out here. Keep these tips in mind when deciding what should be your first foray into the world of Game Boy development.

- # Keep It Small and Simple You may want to create a massive open world, or the next Final Fantasy, but as a first time developer you're going to have many hurdles to overcome that will make ambitious and lofty games more challenging to complete. Think of how you can reduce the player's experience to the bare minimum for success. Create just one level or map. Complete a small section of your larger idea. Build the "Demo" for what will be a larger game later or just make a short game. Once you've successfully built and released a small game, you'll have the tools to tackle something larger at a later time.
- # Replicate A Proven Concept It's no surprise that many games are built on the concepts and ideas of games that came before them. While it may seem cheap or unoriginal to copy an idea, knowing what mechanics and design choices work from previous games will help you focus on how to actually pull them off. Most great artists and musicians got their start by trying to replicate another's work, why should game design be any different?
- # Have A Plan It's tempting to go full throttle into building your game right away, but not knowing where you are going can lead to dead ends and frustration. Write out your game's "script" if you have a lot of dialogue. Draw maps on paper. Type out "pseudo code" of how you think the mechanics will work in the script. Figuring out where your game is going beforehand and how to accomplish it will help keep track of the bigger picture and not get bogged down in the details. Speaking of which...
- # Build The Gameplay First A not so polished but functional game is far more playable than a pristinely presented game that's missing mechanics. So many first-time developers get fixated on the presentation that they neglect how the game actually plays. If you watch the development of any big budget game, the art assets are the last part of the game to be built, with much of the mechanics being prototyped with primitive graphics. It's like building a house, you wouldn't start to decorate and paint the walls before the roof is up.
- # It's All Math Everything in a game boils down to math at some point. If you aren't sure how to achieve a mechanic, look at it from a mathematical and logical perspective. Variables are all numbers being stored, so how can you manipulate and compare those numbers to achieve your goal? If a script isn't working or achieving a desired result, chances are the math is wrong or you need to break down the steps further into their simplest form.
- # Get Help and Input There is a wealth of knowledge out there from the many other games and developers who've been using GB Studio since its launch in 2019. Call upon them for guidance on how to achieve the results you're looking for. Read the docs (or this site), check out the reddit page or join the discord. Go look at the many GB Studio tagged games on itch and contact the developer.

See if they’ re willing to share some insights. Bounce ideas from your friends and people outside of the community. Seek collaboration on the art and music from those who may be more skilled than you, or ask someone who is good with coding to help figure out the scripts you need for your game. When you have a free and open source tool that is designed to be easy to use, you’ ll find a myriad of people who are willing to help you use it. # Take Breaks If you are stuck, stop and take a break. Go have a coffee, go for a walk, listen to music or read a book. Do anything that takes you away from the computer. Sitting and staring at a screen can be detrimental to both the development of the game and your health. Taking a break will help recharge you, and you’ ll be surprised at how easily some things come once you take some time away from the project. # If All Else Fails... Go to the shop and make breakfast. No, not in real life – make a game where the character has a shopping list for items needed to make their breakfast (eggs, milk, cereal, etc.) and build a small multi-scene world where the goal is to get all the ingredients, then go home to cook it. This simple scenario should get you all the basics for working with scenes, sprites and actors, variables and simple scripts. Once you’ ve built this, you should have a taste for the GB Studio workflow and can move on from there.

II. Development Workflow & Design Principles

- Create A(n almost) Seamless Transition
- Dealing With Burnout
- Development Workflow - Chapter 1: Researching Your Game
- Development Workflow - Chapter 2: Brainstorming Ideas
- Development Workflow - Chapter 3: Prototyping your Game
- Development Workflow - Chapter 5: Discipline & Demotivation
- Development Workflow - Chapter 6: Readability (Part 1)
- Development Workflow - Chapter 6: Readability (Part 2)
- Development Workflow - Chapter 7: Color Theory
- Development Workflow Chapter 10: Working As A Team
- Development Workflow Chapter 4: Level Design (Part 2)
- Development Workflow Chapter 8: Replayability
- Development Workflow – Chapter 9: Creating a Color Palette (Part 1)
- Development Workflow – Chapter 9: Creating a Color Palette (Part 2)
- Dissecting Classic Cutscenes From the Game Boy
- How to Make a Responsive Action Platformer
- How to make a Game Boy SHMUP with GB Studio without losing your mind
- Making a Cutscene for the Game Boy in a Week
- Solving the 'A Button Paradox'
- Ten Lessons I learned During My Latest Project with GB Studio
- Top Five Mistakes for First Time GB Studio Users
- Zen and the Art of Retro Level Design in “Kudzu”

Dealing With Burnout

It has unfortunately become more prevalent in the creative fields – especially in game dev communities, so it’s time to talk about burnout, a very real and potentially very dangerous issue that has plagued many of my friends and affected me personally. # **What is Burnout?** Burnout can take many shapes, but generally it’s when you feel physically, mentally and emotionally exhausted. This is usually caused by excessive and prolonged stress and can make someone have reduced motivation and lower productivity in both their work and everyday life. You are left feeling lost, hopeless and resentful. Burnout can also manifest physical symptoms like back aches, headaches and loss of appetite. All this can compound to make you lose sleep and feel generally unwell. Burnout also causes people to withdraw from their friends and loved ones making recovery extra difficult. This is all very real and has been researched and documented by national mental health organizations like CAMH. # Dealing with Burnout As the saying goes, an ounce of prevention is worth a pound of the cure. There are many ways to keep burnout at bay, but the key thing to acknowledge is it’s not explicitly a matter of how much you work. There are many individuals who work longer hours and deal with their stress without issue, and there are those who can’t muster more than a few hours of intensive work a week due to the impact it has on them. What works for one person may not work for another, so keep that in mind when trying to figure things out. These suggestions can help prevent burnout, but also work at mitigating while being impacted by it. ## Eating Well Becoming over stressed with work demands can cause you to neglect what you’re putting into your body, even sometimes forgetting to eat altogether. A healthy diet that’s comprised of mostly fruits and vegetables will net a positive impact in many areas of your life, but also help your mental health and state of wellbeing. Keeping hydrated is also important. ## Exercise This may seem reductive considering the previous suggestion, but once again regular exercise helps keep the body and mind healthy and can help combat burnout. This is especially true in the game dev world, as our jobs often require being sedentary for long periods of time. It doesn’t take much, 20 minutes of light activity like walking a day is enough to make a significant impact. It also helps break the monotony of the day and if you’re able to get outside can brighten your mental state. ## Work/Life Balance This is what it mostly comes down to. If you are working constantly and don’t give yourself time away from your projects, you’re setting yourself up for burnout. Spend time with friends or family, or socialize with your coworkers on the job. Social support is the best way to combat burnout, although with the pandemic it’s been difficult to maintain those relationships. Read a book, watch a movie, listen to music, play a game, do anything that takes you away from your project and nourishes the other parts of your life. You may want to “ride a wave” of anticipation for your game and work quickly to release it, but games take a long time to create and refine, even with an easy to use tool like GB Studio. Which brings me to my next thought: ## Manage Expectations The Game Boy was wildly popular and had one of, if not *the* longest lifespan of any console, but it was released

more than 30 years ago. Games, game development and the gaming community at large have long since moved forward and don't focus as intently on playing games for the Game Boy. What you are creating with GB Studio should be seen as a hobby first and foremost, something that is a form of self expression and a creative outlet. If you are able to bring in some financial benefit, it's a bonus. Yes, we've seen some wildly successful modern Game Boy projects, but those games are an exception, not the norm. Setting expectations of creating a "hit game" is one sure way to set yourself up for disappointment, pushing your motivation and mental state down further. Even modern indie games are a hard market to succeed in, ask any developer who's released work on Steam, Itch or on Consoles. This article itself is an example of putting life before production. It was slated to be released several days ago, but I have prioritized the other areas of my life that needed focus. This has been the culture that I've tried to maintain at GB Studio Central from the beginning. It is a community based project, and contributors are able to submit articles when they want – not when the business demands it. No game or project is worth putting your health in jeopardy. In the end, it's just a game. It should be fun and if producing it is detrimental to your health, where's the fun in that? # What If It's More? It's important to recognize that prolonged exposure to stress can lead to more severe mental health issues. Last year after months of dealing with the effects of burnout, I developed Generalized Anxiety Disorder. This is largely caused by chemical imbalances in the brain that result from the body trying to deal with the effects of stress hormones. In order to get better, I needed professional help which included medication, which I am still taking. This is the message that I want to relay to our readers, too many times we think that what we're dealing with is something trivial and will either subside on its own or it will be a burden to others and not worth seeking help for. Your friends and loved ones want you to succeed, and asking for help is how it will get better. You may not need a course of therapy to counteract the effects, but you won't get better unless you talk to someone about it. If you are concerned that you are experiencing burnout or other mental health challenges, reach out to a loved one or a friend, and/or speak with a medical professional. We're all in this together, and these last few years have been extremely detrimental to the overall mental health of a large portion of the population. Your game can wait, we'll be here to see it. Emi Paternostro

Emi Paternostro

Audio Engineer, Mac Technologist and Video Game Developer. Managing Editor of GBStudio Central. (he/him)

proximitysound.com ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn

- [Click to share on Reddit \(Opens in new window\)](#) [Reddit ### Related](#)

Development Workflow - Chapter 1: Researching Your Game

So, you've got your next big idea for a new Game Boy game! Maybe you were playing another game and you found a mechanic worth exploring more, or an idea just popped into your head out of nowhere, or you just always dreamed of making the next big Metroidvania or RPG for the Game Boy! One thing is certain, you're excited and want to jump right into development. Hold on there you eager beaver. While it's tempting to run off to the land of code and get stuck in, I'm here to say there is an important step that comes before prototyping your next project – Research. Researching your game can maximise your ability to brainstorm ideas effectively and minimise running into problems later in the development cycle. There have been many times when I have had to scrap hours or days of work because I didn't foresee an issue and had to redo aspects of my games design as a result. By researching other games, you will be able to benefit from the experience of other talented games developers who have almost certainly solved the problems you will face during the development of your next masterpiece. Standing on the shoulders of giants is far better than standing alone. So where do we begin? With a search engine. It's time to discover games that share commonalities with your own. For the purposes of this article, we will imagine our next project is a puzzle platformer. A quick search has uncovered a slew of inspiring games from the GB and GBC library. Here are a few we can consider:

Clockwise from top-left: Donkey Kong 1994, Bugs Bunny: Crazy Castle Series 1999, Sutte Hakkun GB Unreleased, Toki Tori 2001 We have a list of games from which to begin our research, so what's next? It's time for the fun part, playing games. If you can't get your hands on a copy of the game you want to play, you can always watch a 'let's play' on YouTube and record your findings from that, but it will be more difficult to understand how the gameplay *feels* if you don't play it yourself. While experiencing the game, you will need to describe what you see. And that means one thing, Spreadsheets, spreadsheets, spreadsheets. If you don't already have spreadsheeting software, there are some free open source options such as OpenOffice 'Calc' readily available online, and Google Sheets is an excellent web based tool that can be used for this purpose. Start by opening a new document and labelling each sheet by Game (down the bottom left in Calc and Excel). This way, you will have your information organised by game in one easy to access file. For the purposes of this article, we will work on only describing Bugs Bunny: Crazy Castle 3 on GBC, but it's up to you how comprehensive you wish to be during your research phase. Next we need to start describing each mechanic or feature we come across while playing. You can organise columns/rows within your spreadsheet any way you wish but I recommend including the following information as a start: 1. *The Name of the*

mechanic/feature. 2. *A screenshot of the mechanic/feature*. It is often easier to visualise mechanics with images rather than words and a screenshot will jog your memory when referring back to these notes later on. 3. *A detailed description of the mechanic/feature* 4. *Whether you think it is possible to recreate something similar in GB Studio* (while this point is not critical at this stage, it doesn't hurt to think about how aspects of your game may be coded) 5. *Any other relevant notes*. For example: Initial thoughts on coding the mechanic in GB Studio and most importantly – how the mechanic made you feel during gameplay (More on this in a moment). Feel free to organise your findings in any way that suits your needs, you can add to (or remove) these dot points as you see fit. Using Bugs Bunny: Crazy Castle 3 as our example, our first mechanic could fill out the five points above as follows: If you are unsure as to whether coding a mechanic is possible in GB Studio, you can always head to the GB Studio Discord and ask the community if they have any ideas. Furthermore, when writing notes regarding a specific mechanic, try explaining how it makes you feel. What aspect of the mechanic made it fun? Did it have something to do with how it was animated, or the sound effect that was associated with it? On the other end of the spectrum, perhaps something just felt off about a certain mechanic – I never liked the feeling of bonking my player characters head on ceilings in platformer games, for example, it just doesn't feel good. Why? Because I have lost all my momentum all of a sudden. If aspects of a mechanic anger or frustrate you over time but it's otherwise an interesting one, the mechanic may benefit from a few tweaks when applying an iteration of it in your own game.

Your spreadsheet may look something like this before long – but don't stop there, these four mechanics and one feature are only the tip of the iceberg. Having gone through a number of games, documenting their various successes and failures and creating a robust research document of your own, your head will be positively buzzing with ideas of your own and you will have laid the groundwork to make the best game you possibly can. The games you select for the purposes of research need not be exclusive to the various Game Boy consoles. You will find valuable information that can come from any game. Maybe the level select menu can be a predetermined grid based path akin to Super Mario Bros. 3 on the NES, or a free-roaming map like the level select screen in Super Mario 3D World on the WiiU/Nintendo Switch. If you want to get really lateral with your research, you can even explore ideas from table-top and board games. In Chapter 2, we will explore the next step in the development cycle: Brainstorming Methods.

This article is part of a series: Development Workflow Gumpy Function

Gumpy Function

Independent Games Designer, Artist, Film Enthusiast and Full-time Dad (he/him). Check out my games here!

www.gumpyfunction.com/ ### Share this:

- [Click to share on Bluesky](#) (Opens in new window) Bluesky

- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### *Related*

Development Workflow - Chapter 5: Discipline & Demotivation

Can I start by asking a hard question? How many times have you started a project and lost interest at some point down the line? If you are anything like me, it's more than you can count. I have hard drives filled with artwork, sketches, sprites, animations, music, sound effects, stories, plans and much much more that have never featured in a fully realized game fit for public release. If this is also the case with you, please don't be ashamed or upset in any way. I would consider this practice a right of passage for the budding game developer. Only by failing multiple times can we learn and improve upon ourselves bit by bit and build on our personal knowledge to the point at which we can finally release our first *finished* game! Here is the thing though, I'm not talking about any old knowledge. I'm not talking about learning the in's and out's of coding in GB Studio, or the skills necessary to draw a great sprite or background or how to write a great piece of music or story. I'm talking about learning about yourself. About your own limitations as a person and about what works best for you when it comes to your own workflow. I touched briefly on discipline in Chapter 3 – Prototyping your Game but I'd like to explore this subject in greater detail as well as some additional mental challenges the development cycle never fails to throw at us. Specifically, we will be discussing the issues of self-discipline, de-motivation and one more thing I'll save for the end. These are extremely important subjects; as the ways in which we approach these challenges will often mean the difference between completing or abandoning our projects. While learning about how you work as a person is largely a private journey, we can still explore strategies that will assist us with building self discipline and working through periods of de-motivation together. ## **Feeling Demotivated is Inevitable** Experiencing feelings of de-motivation is inevitable in all projects worth doing. That sweet honeymoon phase where you seem to have unlimited energy at your disposal will soon vanish and some aspect of the project will turn into *work*. Whether it's some troublesome coding issue, or a tedious play-test session or otherwise, it's just a fact. The sooner we accept this, the sooner we can work on overcoming this problem. As I mentioned earlier, this is a personal journey. So I thought it best to list a whole lot of ways to get through a rut, a kind of shotgun approach if you will, and leave you to try out whatever you think may be applicable to your own sensibilities. What works for one person, may not work as well for another. Furthermore, what works for one person *today*, may not work for that same person *tomorrow*. 1. **Work for just 5 minutes**,

even if you limit yourself to making a single thing. Draw a single sprite, write a single line of code... anything! The simple act of starting may snowball into something positive and you will be back in the flow of it. But if not, don't fret, you can try another five minute stint tomorrow. Which brings us to... 2. ***Take a break!*** Sometimes I have burnt out from spending too much of my time on a single project. Working at a breakneck pace is unsustainable so don't forget that some fresh air or switching to another project may be just the thing you need. 3. ***Play Some Games.*** Sometimes, all you need is to reignite the passion. Try playing some games of a similar genre to the one you are developing. At worst you will have a nice time enjoying a great game, at best you will find the creative juices flowing once more. You may even find yourself adding to your research document or naturally brainstorming ideas for your own game. In that case, you're already back in development. 4. ***Pivot your Idea.*** It can be depressing working on an aspect of a game that you feel uninspired by. Perhaps you were inspired at some point, but the realization that what you are building just isn't as engaging as you initially hoped sets in. If that's the case, admit it and begin re-focusing towards something that does excite you. Try to pivot your idea rather than scrap it entirely. 5. ***Cut Content*** if you find yourself becoming burdened under time constraints or heavy work loads. Sometimes the perceived mountain of work in front of us may cause us to feel overwhelmed and push us towards procrastination. Lightening the workload or breaking it up into smaller, more manageable chunks may help to reduce this. Be utterly ruthless if you have to, you can always expand the scope of your project later if you are feeling confident enough. 6. ***Shift work to another aspect of Game Design.*** This will get your head into a new space and remove it from the space it was struggling with. If you are bashing your head against designing levels for example, switch to writing the story, do some artwork or even designing mechanics – then you can come back to level design with new ideas and the mental block may have disappeared. ## **Self-Discipline is a Learned Skill** No one is disciplined straight out of the gate. Of the dozens of games I have started, I have completed only a handful. It took me years (and years) to finally complete a game I was happy with and felt I had explored to the best of my ability. Like a game of Tetris, all I could see was my mistakes piling up. This is a counter productive way to look at it though. Instead, I realized that these failings were actually a road that would ultimately lead to success if I just kept trying. So here are a few ways to help you stay on that path to victory. 1. ***Keep things small (at first).*** A small scope will make it easier to finish a project. As a beginner, starting with a large open world RPG for example, is simply too much to expect from yourself. You will learn far more by completing several bite sized games rather than one huge project – especially if that project is never completed. Slowly expand the scope of each project as your confidence in yourself and your ability grows. 2. ***Enter Game Jams.*** By definition you will be completing small projects and releasing them for public consumption. This means you will be getting better at completing aspects of your games that you are not particularly motivated to do or are good at (at first). It's easy to get stuck in to the artwork if that's your main passion, it's another thing to do all the coding or build levels and so

on. Keep at it, the more projects you complete, the easier it is to work on what you consider to be the less interesting aspects of game design. 3. ***Collaborate!*** If you really want to focus on a few aspects of game design (or even just one), then that's fine too. Why not collaborate with others in that case? Working with other developers is also a great way to stay motivated because you can feed off each other's enthusiasm. 4. ***Take your time and plan.*** Pacing yourself and taking the time to properly plan will help break the development cycle into more manageable chunks. It's easier to commit to work when you know you need to climb a hill that day, rather than a mountain. 5. ***Explore other hobbies too.*** I remember a time when I couldn't finish an oil painting no matter how many tiny canvases I started. It took patience and persistence to get to the point where I am now, and even then I am still trying to finish painting something larger than a couple of square feet. Setting goals outside of game development and completing them will go a long way no matter what you do – it's all building on this one skill. 6. ***Know when to quit.*** Ultimately, you don't want to end up forcing yourself to do something you just plain don't want to do. Do your best to understand your own limitations and what caused you to lose interest. But most importantly – don't be too hard on yourself. If you need to move on from a project, then so be it. ## **The Developer Blues** The last thing I want to touch on in this article is something that we, as indie developers, struggle with perhaps the most. And that is what I'm going to call *developer blues*. It takes a lot of work, patience and perseverance to finish a game. And one of the most upsetting aspects of the indie developer struggle is seeing our hard work go unnoticed. Exposing our games to a general audience can be very difficult – particularly with something as niche as Game Boy development. We spend countless hours pouring our heart and soul into something we care deeply about and sometimes, no matter how hard we market ourselves and our projects, it can seem like a hopeless enterprise. I have had times when I cannot understand why my games aren't getting the attention I believe they deserve, and I know others feel the same way. Even those who I feel have seen the most success and who produce the highest caliber of work in the Game Boy scene, they too experience the developer blues. It's important to manage our mental health when developing gets us down, so I'd like to spend a moment discussing some strategies that will help when we are feeling depressed. 1. ***Talk to other developers*** when you need support. Your struggle is shared by many, and I'm sure you can find some words of wisdom or good old positivity when you really need it on the GB Studio discord. 2. ***Share the work of others.*** What better way to curb the blues than do something good for another developer. Why not sing the praises of someone you admire or someone who needs some exposure of their own. The indie scene doesn't need to be a competition. We are all in this together! 3. ***Stop counting numbers.*** Twitter points, itch analytics, dollar signs and all those sorts of things don't matter in the end. All that matters is you are enjoying the process of Game Boy development and expressing yourself creatively. 4. ***Get flashy!*** It's often best to share the more exciting aspects of your game. A great splash screen, or a single highly detailed cut-scene image (or GIF) will often grab the public's attention more than a complex show of game

play. Spending time on the more aesthetic aspects of game design is a great way to blow off steam as well. 5. ***Manage your expectations.*** Above all else, this one is key. Your work is important and well worth doing. Under-exposure can hurt but be patient, and keep going. Eventually you can get there – especially if we offer support to our peers. How we apply motivation and self-discipline strategies and even our attitude towards developer blues tend to crossover much of the time because they are all tied to the same issue. That is, understanding how we as an individual operate. The process of trying, failing and trying again is just a pathway to your very first finished game or, if you have already achieved that amazing feat, a pathway to that dream game you always wanted to make. In any case, it’s up to us, as individuals, to understand how we think, when to start, stop, keep going or slow down. Only then can we build on our mistakes and complete what we set out to achieve. In the next Chapter, we will shift gears back into practical game development, and discuss how best to present visual information to the player – the subtle art of readability.

This article is part of a series: Development Workflow Gumpy Function

Gumpy Function

Independent Games Designer, Artist, Film Enthusiast and Full-time Dad (he/him). Check out my games here!

www.gumpyfunction.com/ ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### *Related*

Development Workflow - Chapter 6: Readability (Part 1)

Have you ever heard of a game called *Maru’s Mission*? If you haven’t, you are among the vast majority. It’s a largely forgotten action platformer that stumbled onto the Game Boy back in 1991, and I do mean stumbled. It has… problems. It’s quite buggy and the player character controls are just baffling! But its biggest issue, at least to me, is its poor *readability*. Let’s take a look at a screenshot and I’ll explain what I mean:



Maru's Mission 1991 That's Maru the ninja, standing on the tree stump. As you can see, the tree stump is a solid object acting as a platform. Maru can stand on it and, when he walks into either side, Maru will stop in his tracks. So what about the large tree trunk to Maru's right? It's using the exact same repeatable graphic as the tree stump so, logic would imply, the tree's trunk would also be solid, right? Well, no. Not in this case. That tree is, in fact, a background element. The tree trunk *looks similar but is functionally dissimilar* to the tree stump, and that just doesn't make much sense! What we have here is a readability problem. In other words, *a visual element is not clearly distinguishable relative to another visual element*. This is just the tip of the iceberg with this particular game's readability issues. When Maru steps in front of the tree for example, his body is a similar tone to that of its trunk, making it very difficult to see him as he passes in front of it. The enemies have similar issues as well, making them hard to see at times. And shifting the color of the sky towards the lighter end of the spectrum would certainly improve readability too. Now let's take a look at a game that is an example of excellent readability:

Super Mario Land 3: Wario Land 1994 What is and isn't a solid surface is

so much more clear here, isn't it! So what exactly makes *Maru's Mission* so disastrously unreadable while *Wario Land's* graphics make it clear what is going on at a single quick glance? Well, while it's true that the latter certainly has a more aesthetically pleasing art style, pixel art skills don't actually enter into it. What's important here is the artist's understanding of *tone*. Even the most beautiful pixel art assets will lead to a frustrating player experience if tone is not properly managed during art asset design. So even if you feel your artistic talents are lacking, don't fret! As long as you have a handle on tonal differences from one game element to the next, the end result will work out just fine. Let's explore *tone*, one of the pillars of art, and what it means within the context of game design. *Tone is a measure of the lightness or darkness of a particular color.* For games developed specifically for the DMG, this aspect of decision making is greatly simplified for us, as there are only four tones of a

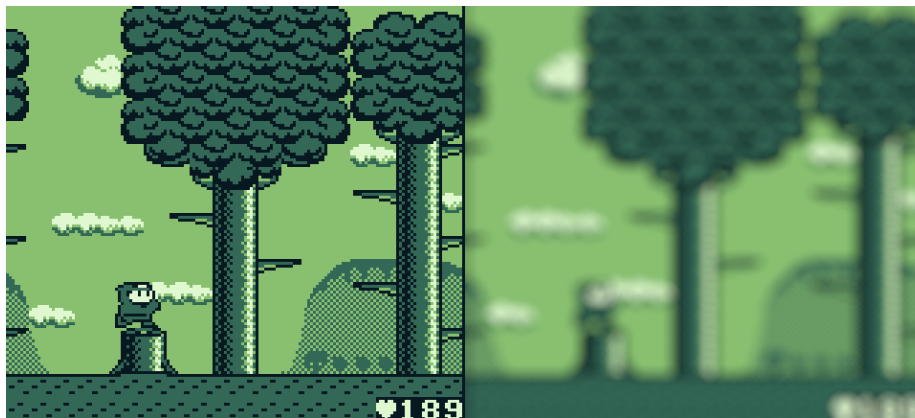
single color to choose from!



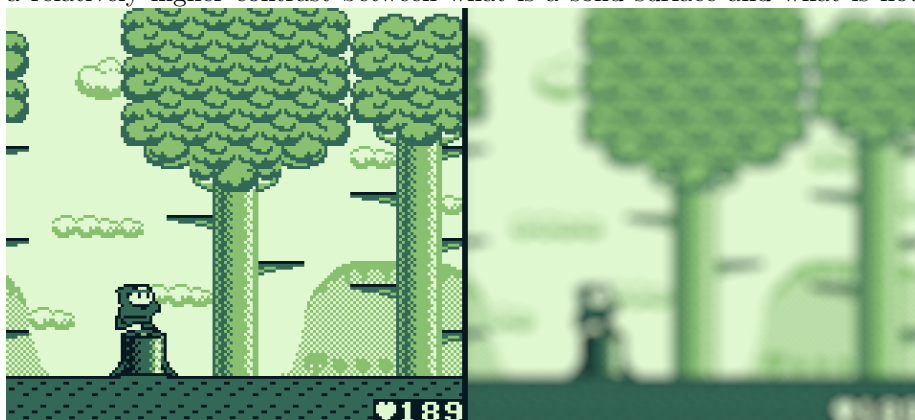
White, light green, dark green and black. The four colors utilized in DMG mode in GB Studio. Whether you are creating an RPG, puzzle game, platformer or otherwise, making sure the elements that make up your game are visually distinguishable from one another is a must if the player is going to be able to navigate the game's space easily and without confusion. The areas in which the player can move freely and the adjacent areas that are impassable (like a solid wall for example) are best separated visually by creating a strong contrast between two tones. This goes for separating interactable objects/NPCs or enemies too. When we are talking about separating elements visually and managing readability using tonal differences, what we are really getting at is this: *Creating high contrasts between one element and another makes information clear and easily readable.* One way to test whether one area of the screen (for example, the background layer) is contrasted appropriately against another area of the screen (for example, the platforms on which a player can stand) is by conducting a *squint test*. The Squint Test process is as follows: * Design your background art tile set, enemies, interactable objects etc. * Lay them out in one single image so that it acts as a mock-up of a potential level. * Squint your eyes until most of the detail is lost, and all that is left is a vague suggestion of light and dark areas. By doing this, you can assess whether your art assets are easily readable in relation to one another *without the details of the scene clouding your judgment.* To simulate this, we can blur the two examples from above. (Blurring the image using software is another fine option if you would prefer not to squint.)



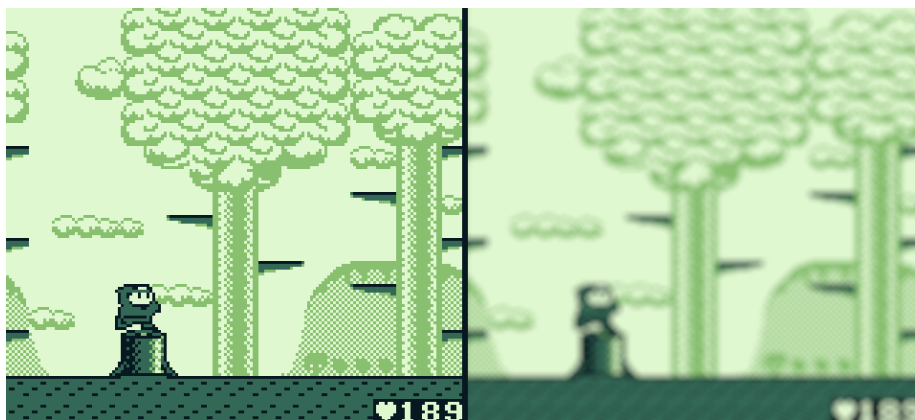
The scene from *Maru's Mission* (left) and *Wario Land* (right) using the squint test/blurring the image. By removing most of the detail in the image and focusing solely on areas of tonal contrast, we eliminate the rational part of our brain that attempts to *contextualize* the space beyond simple light and dark spaces. For example, if we focus on the blurred image of *Maru's Mission*, we can see there is a very dark vertical line that acts to 'block' the path forward from left to right. We know that, when the image isn't blurred, that is a tree. And if we were to play the game, our brain would get to work on it. It would say to itself: "Hey, I know that tree trunk is dark just like the stump and, although the stump is a solid platform, the tree trunk is just a tree in the background. Therefore, I can walk past it." But here's the rub, that thought process takes time – even if it's just a millisecond. The brain is very good at making sense of things and it makes sense of the game's landscape eventually but, when we are in the heat of the moment (and certainly when playing an action platformer), that's milliseconds of time wasted when the brain should be focusing on more important game play elements. You know, things like judging the trajectory of an oncoming projectile or where the player character is going to land when jumping. Time spent making sense of a background element might mean the difference between reaching a win state or a fail state so it's up to the developer to make sure the visual information available to the player is as clear as possible *at a glance*! The "*at a glance*" part is *really* important. If the player can make sense of the playable space within a scene almost instantaneously, then you will have succeeded in solving the readability problem. *Wario Land* is always making sure the background has high tonal contrast relative to its platforms, collectables and enemies and the end result is, well... one of the best games on the system! Let's apply what we have learned so far to the *Maru's Mission* scene and see if we can "fix it", so to speak. Here is the original, in all it's unreadable glory:



Maru' s Mission in its original form. Next, we have the same scene but I have shifted the tone of the background elements to a lighter color, creating a relatively higher contrast between what is a solid surface and what is not:



Maru' s Mission with a re-evaluation of tone. Notice I have left a black line on the top surface of the tree branches. They are one-way platforms that the player can land on, so I have attempted to convey their similar function to the ground platform by having them retain their dark tone. Furthermore, the sky and distant mountains in the background have been shifted to a lighter tone as well, which helps create a high contrast between the empty space and the solid ground. Lastly, if we want to push the contrast as much as possible, we can shift the tone as follows:



Maru's Mission with a re-evaluation of tone but this time taken to the extreme. Using the Squint Test, it's certainly clear what is and isn't solid now – especially at a glance! In addition to sorting out our readability problems, changing the tone in this way has also conveyed a sense of receding dimension too. Particularly in the first iteration, the background elements have been separated to create a foreground with the player character, tree stump and ground, a mid-ground with the trees and a background with the clouds and mountains off in the distance. So tonal variation can also do more than simply make the game play more readable. This exercise has made one thing abundantly clear; Just because something is dark (or light) in the real world, *doesn't mean it should be the same when recreated in a game*. I can imagine the artist that worked on *Maru's Mission* may have had a thought process akin to the following: “Tree trunks are dark brown, so therefore the tree trunks in the game will need to be a dark color. And certainly, since I'm making the tree stump dark green, the trees need to be dark too because they are made of the same stuff!” This kind of logic is appropriate for realism within the realm of fine art. But when applied to artwork in video games, true likeness can often get in the way of how the player interprets the moment to moment game play. Ultimately, readability should always trump realism when it comes to designing visual game elements. Otherwise, there is a risk the game will become very confusing for the player – and they may not even be able to put their finger on why at times. Confusion leads to frustration, which leads to drop off or even the dreaded rage quit ... and we certainly don't want that for the player! Before we finish up, there is one more way to use strong tonal contrasts to our advantage that I'd like to briefly touch on. Whether it's a player character or enemy in a platformer, or an important item or object the player needs to interact with in an RPG, creating a *bold line* around the point of interest will increase readability drastically and direct the players attention towards what is pertinent to game play. A bold line can mean a black line against a white background, or a white line against black background, or just a relatively thick line of any tone. On the flip-side, just as elements with a lower detail don't grab our attention relative to those with lots of detail, likewise, by reducing the

contrast of art elements within the background layer that are *not* important to the game play, but serve only as “set dressing”, we can lower their visibility too.



Toronbo Shores. The Legend of Zelda: Link's Awakening 1993 The coconuts featured in *Link's Awakening* are a solid obstacle so feature a black outline to indicate this. The seashells in the sand on the other hand, serve as background detail only. They are “pushed back” into the background by only utilizing the three lighter tones and omitting black all together. In fact, this scene is a prime example of all we have discussed in this article. All the interactable objects and solid barriers use a black line and relatively dark tones to separate the non-playable space from the playable space (which itself only uses the two lighter tones available). The eye is drawn to the detailed, high contrasted treasure chest in the top right. And the camera stays put while the player navigates the space, only shifting to a new view when moving to a new map screen. Great stuff from Nintendo's Research and Development No. 4 department, as always! In Part 2, we will continue looking at the issue of readability. Specifically, we will explore how dark tones can negatively impact game play on the dot matrix screens of the DMG and Game Boy Pocket.

This article is part of a series: Development Workflow Gumpy Function

Gumpy Function

Independent Games Designer, Artist, Film Enthusiast and Full-time Dad (he/him). Check out my games here!

www.gumpyfunction.com/ ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### *Related*

Development Workflow - Chapter 6: Readability (Part 2)

When I was 10, I remember playing *Donkey Kong Land* when it was first released back in 1995. I was a big fan of the SNES version that had exploded onto the market a year earlier, and was excited to see Rareware's take on Donkey Kong barrel it's way onto Nintendo's handheld console. When I turned on my Game Boy and began playing stage one however, I was alarmed to find that the large sprites were quite hard to see against the detailed scrolling backgrounds! So much so, that I found myself losing life after life, trying my best to make sense of the muddy tones of green that wildly smeared across the screen as I played. The developer had faithfully translated the artwork of *Donkey Kong Country* as best they could (minus the color, of course) but that choice made it difficult to see platforms, enemies and even Donkey or Diddy Kong as they moved across the dot matrix screen of the original DMG. Even the Game Boy Pocket, with its improved display, also succumbed to the debilitating problem of blurred visuals when anything was in motion.



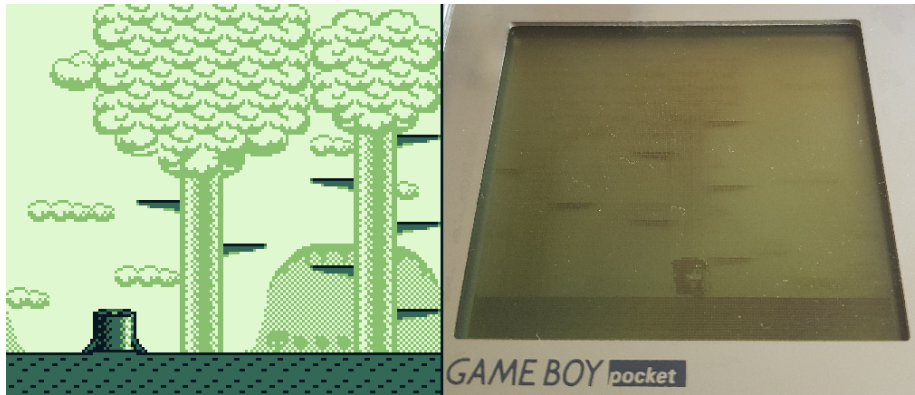
(A vague suggestion of) *Donkey Kong Land* on the Game Boy Pocket. You have to admit, it's not easy playing that game on un-modded hardware... and it's considerably worse on the DMG. These days, we are spoiled with backlit, high resolution screens packed into our portable devices that would have ten year old me weeping with joy. But, back in those days, no one was the wiser. I saw a new portable Donkey Kong game and, without question, bought it (and loved it), warts and all. The fact remains though, if you want a game to be readable on original hardware, the artwork needs to be very carefully considered! And I'd be remiss if I didn't cover this subject for the purists out there who love to play their Game Boy games as they would have originally been presented back in the 80s and 90s. In Part 1 of this article, we explored *Maru's Mission*, an action platformer released in 1990 by Jaleco with some unfortunate readability issues. I spent some time revising the tones in stage one to provide a higher contrast between platforms and the empty space surrounding those platforms, thereby improving the readability during game play. Before we take a look at those revisions on real hardware, let's see what *Maru's Mission* looks like in its original state for the sake of comparison:



The first level of Maru's Mission 1990 as seen on the Game Boy Pocket screen during play. It certainly looks like the darker tones are causing a lot of the scrolling image to blur, making it extremely difficult to determine what is what! It's still playable of course, but I'm not sure the fun to frustration ratio is weighted in favor of a particularly good time. Let's see what we can do to improve the fun factor... For the sake of this article's demonstration, I took the "re-toned" scenes I created in Part 1 and inserted them into the platformer engine of GB Studio. *Maru's Mission Re-toned* is available to download if you wish to play the ROM on real hardware and see the revisions for yourself. To start, we'll look at the night-time scene, in which the background has been set to the black tone (and Maru's sprite has been colored white so we can actually see the player character):



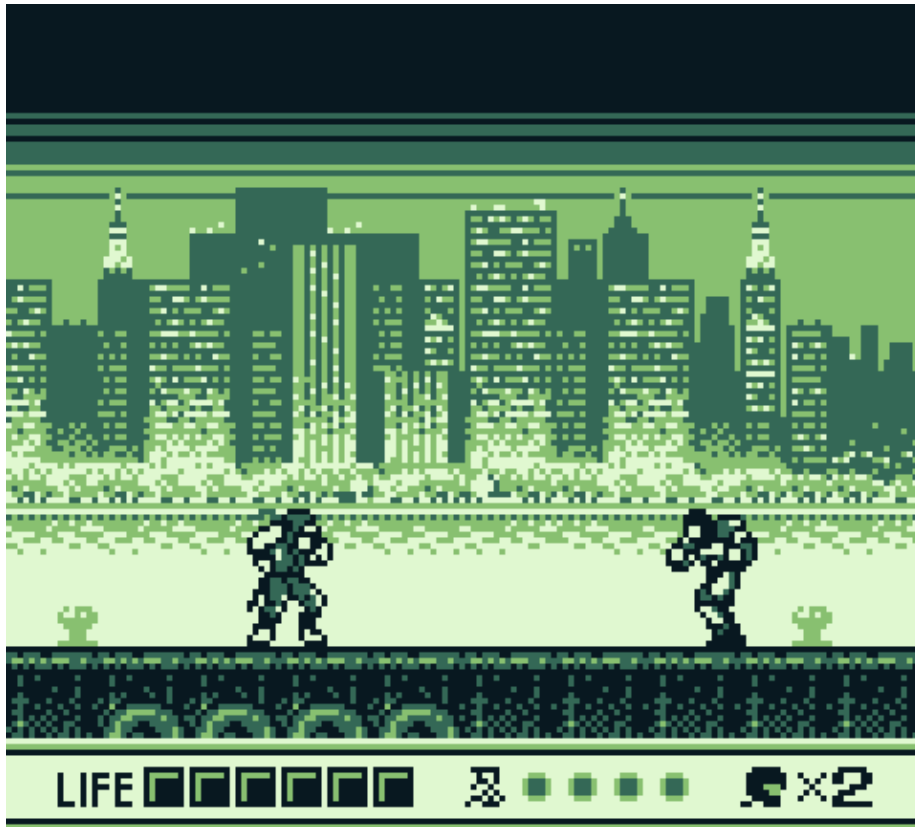
The night-time scene in Maru's Mission Re-toned. The screen looks more like a thick soup than a platformer, doesn't it?! It's becoming very clear the black tone is to be used sparingly if you're developing a Game Boy game with screen scrolling functionality. Now let's take a look at the same scene, but with the background image set to the lightest possible tones while still retaining the tree, cloud and hill background detail:



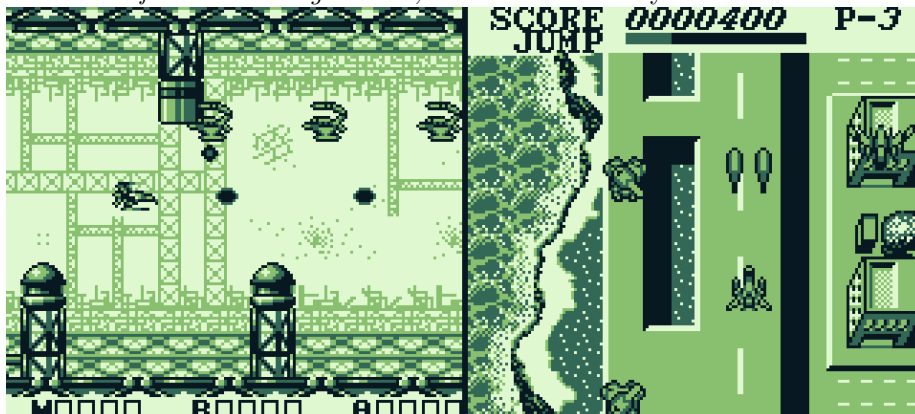
The same scene but with the tone shifted to the lighter colors. The tree branches remain a dark tone to indicate they are themselves a platform, much like the ground extending across the bottom of the screen. The background is using only two tones; white and light green. Now when the screen scrolls from left to right or vice versa, it's much easier to tell the platforms from empty space and the player can more easily focus on the game play rather than interpreting cryptic visuals. But while we seem to be improving game play with our changes, the level of detail in our background artwork is also being reduced as a result! But, by utilizing some well considered art asset layout and limiting the direction of the screen scroll, we can ensure the player can enjoy more detailed and expressive background artwork while still keeping the game play relatively unharmed. We used *Super Mario Land 3: Wario Land* as an example of excellence in readability in Part 1 of this article and we will return to it again here:



The opening screen of Super Mario Land 3: Wario Land 1994 The clouds extending across the top of the background layer of this level as well as the ocean towards the bottom are not only using the light green and white tones to reduce image blur during screen scrolling, but also form clean horizontal lines that extend the whole way through the level. The areas of sand and the rocks at the very bottom of the screen are also constructed in a similar fashion. When the screen scrolls (limited to the horizontal axis only), the details of each art element will blur, but because each is laid out to create a “ribbon” of tone, the game play is not greatly affected. In this way, we can create interesting visuals that can be enjoyed to their fullest ... at least when the screen isn’t scrolling!

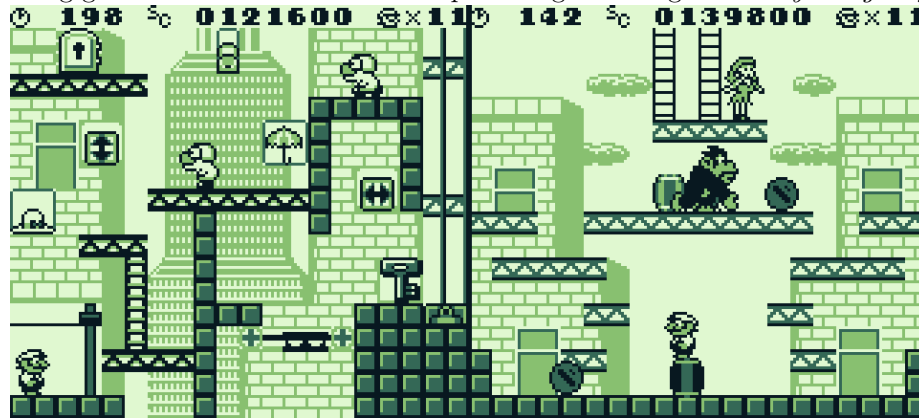


You can even get away with pulling off a night-time scene! *Ninja Gaiden Shadow* 1991 The same strategy can also be utilized for something like a top-down shooter. Basically, as long as the unique areas of detail are constructed to form a clean line and run parallel to the direction of the scrolling screen, decent readability will be maintained.

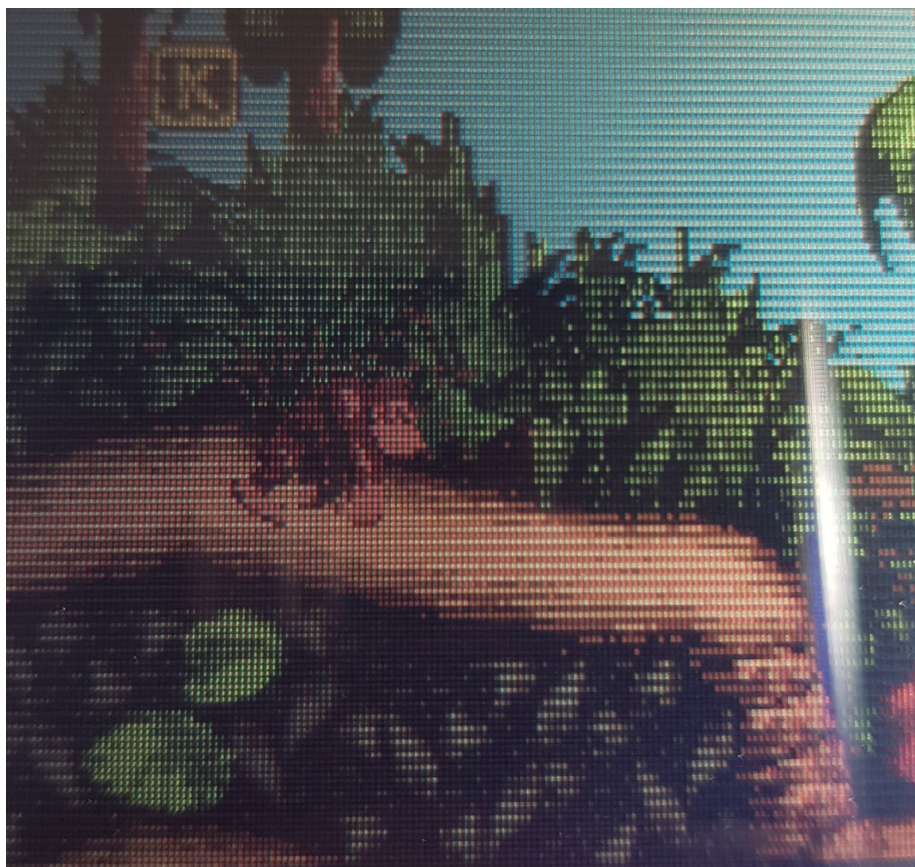


Sagaia 1989, a side-scrolling SHMUP (left) and *Aerostar* 1991, a top-down

SHMUP (right) Ultimately, screen scrolling tends to be a dangerous and risky proposition when developing for the DMG and Pocket. That's why there are so many static screen puzzle games and slow moving RPGs on the system. Even the team at Nintendo's Research and Development No. 1 department knew that the Game Boy is best suited to slow moving platformers. That's likely the very reason *Wario Land* exists as it does, rather than a third *Super Mario Land* featuring the fast moving Mario. A rotund, slow moving Wario, who can't sprint to save his life, just makes far more sense when readability is considered. As a comparison to *Donkey Kong Land*, let's take a quick look at another Donkey Kong game. One that excels in all aspects of game design: *Donkey Kong '94*.



Stage 1-7 (left) and Stage 1-8 (right) Donkey Kong 1994. While not all stages are this small, many limit the playable area to a single screen, removing screen scrolling issues entirely. The detailed background art limits itself to the two lighter tones only to ensure the darker solid walls and steel girders are easily readable when playing. Finally, the enemies, interactable objects and collectible items all have a solid black line around them to make sure they are easily located at a quick glance. *Donkey Kong '94* is made for the Game Boy, and by that I mean it really is a perfect Game Boy game! Thankfully, with the release of the Game Boy Color in 1998, new technology gave rise to new possibilities. Blurry screens became a thing of the past and with the addition of color, developers now had a new tool at their disposal. When applied to the issue of readability, color can be used to push strong contrasts just as tone does, thus improving readability too!



It's still no Donkey Kong Country on the SNES but it's a considerable improvement! Donkey Kong Country (GBC) 2000 as seen on the Game Boy Color. In the next Chapter, we will explore an aspect Game Boy game design exclusive to the Game Boy Color – Palette construction and Color theory! You can find *Maru's Mission Re-toned* available for download [here](#).

This article is part of a series: Development Workflow Gumpy Function

Gumpy Function

Independent Games Designer, Artist, Film Enthusiast and Full-time Dad (he/him). Check out my games [here](#)!

www.gumpyfunction.com/ ### Share this:

- [Click to share on Bluesky](#) (Opens in new window) Bluesky
- [Click to share on X](#) (Opens in new window) X
- [Click to share on Facebook](#) (Opens in new window) Facebook
- [Click to share on Mastodon](#) (Opens in new window) Mastodon
- [Click to share on LinkedIn](#) (Opens in new window) LinkedIn

- [Click to share on Reddit \(Opens in new window\)](#) [Reddit](#) [### Related](#)
-

Development Workflow - Chapter 7: Color Theory

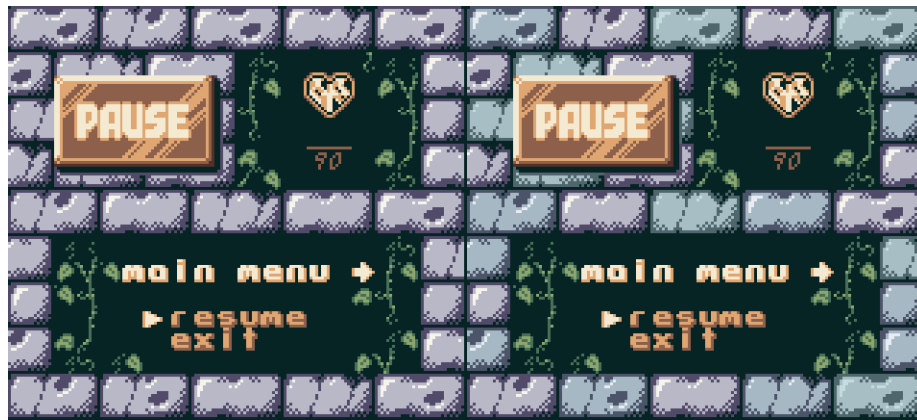
While the Game Boy Color can only display 56 colors screen at any one time, it is able to select those from a range of 32,768 colors, so don't blame yourself if you think palette selection is overwhelming at times! If you decide to enable color mode in GB Studio for your game, then this article aims to break down the decision making process, offer some tips, and explore the various options at our disposal when it comes to all things color. For a in-depth look into how colors are displayed on various hardware, check out SōdoDevs article [here](#). Otherwise, let's get into color theory. If you want a primer or a recap on basic terms then you can read on, otherwise skip ahead to the next section.

Definitions *Hue* – is a gradation or variety of color. It refers to a specific 'slice' of the color wheel or point along the color spectrum. *Saturation* – is the measure of intensity of a color. The lower the intensity the more dull/grey the color will appear. *Value* – is the measure of the lightness or darkness of a color. Here also referred to as *Tone*. *Complimentary Colors* – are two colors that sit at opposite sides of the color wheel. For example: Green and Red, Blue and Orange or Purple and Yellow. *Analogous Colors* – are colors that sit adjacent to each other on the color wheel. For example: Green, Cyan and Blue. Now that the basics are covered, let's get on with it!

Readability using Color In the last chapter, we saw how placing an area of light value next to an area of dark value can create a high tonal contrast between objects in a scene, thereby increasing readability during game play. Colors can also achieve a similar outcome when used in certain ways. *Complementary colors* will draw the eye into areas where both meet, and create a vibrant, lively scene at the same time. For example, by placing a red enemy over some green foliage in the background, it ensures the player will be able to clearly see the danger ahead. When applied to elements such as interactable objects, the developer can signal to the player that attention should be directed towards these items or points of interest. Using *analogous colors* in the same way will make it more difficult for the player to read the environment around them as points of interest may end up 'blending in' too much. This can lead to feelings of frustration or confusion for the player.



A scene from *Uneathed* (2021). Ben Jelter has used analogous blue colors to build the background palette and then applied complimentary colors to the points of interest (namely the pushable rocks, the resource that is required to clear the stage, an optional tutorial on the interactable sign and the player character) in this scene from *Uneathed* (2021). While *analogous colors* won't highlight points of interest such as enemies, obstacles or collectibles in a scene, they can go a long way towards building a more rich and dynamic one. Slight to moderate shifts in hue will subtly transform a scene from looking monochromatic and relatively dull into something that is more striking.



A single palette is used for the bricks in the Pause Menu of *CUPID* (left). However, using multiple analogous palettes for the bricks has livened up what is otherwise a pretty static scene (right). Using saturated colors alongside a similar unsaturated hue is another alternative when creating contrasting areas within a scene. The high intensity of the color will feel like something is ‘jumping out’ or seemingly closer to player while a more muted, unsaturated color pushes itself into the background. By using these properties to our advantage, we can create a sense of depth using a single hue. This is also true when utilizing the difference in *value* of a hue. In this regard, its worth applying the lessons shown in my readability article when incorporating color into a scene as well.

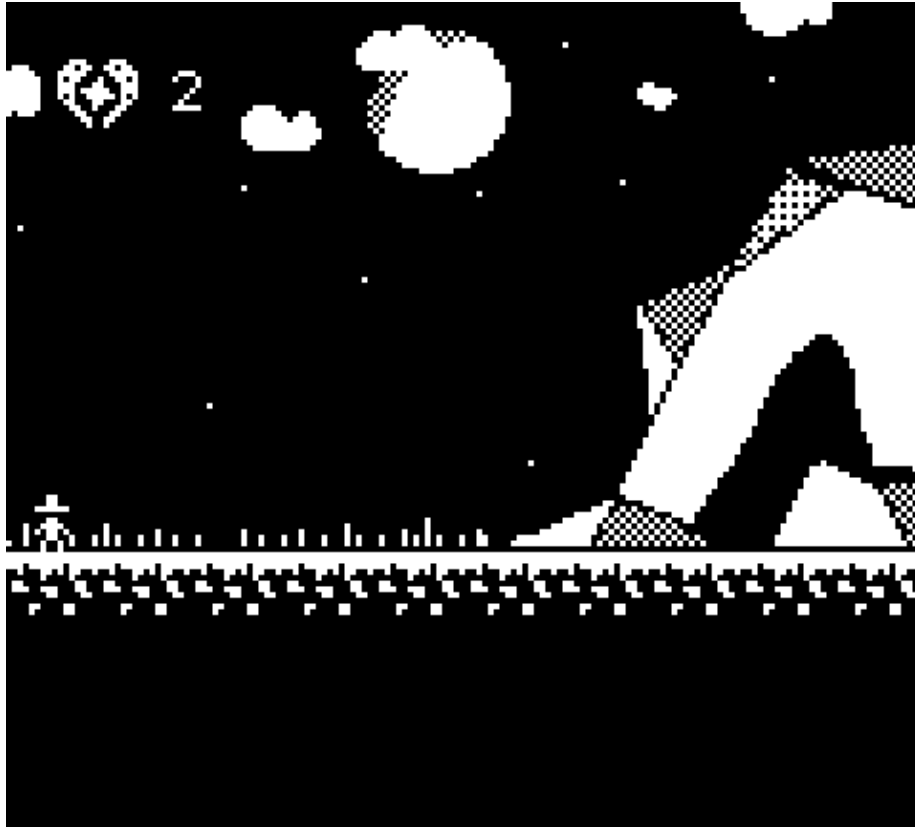
Variation in saturation and value have created a sense of depth in this scene from *Shantae* (2002) In the opening minutes of *Shantae* (2002), the artist has used a very saturated brown for the bridge in the foreground but the brown of the ship in the distant background has been heavily de-saturated. Like wise, so too have the shadow values been pushed to a near black tone in the foreground, but set to a relatively pale dark brown on the ship. Both choices give the illusion of depth (as well as improve readability during game play).



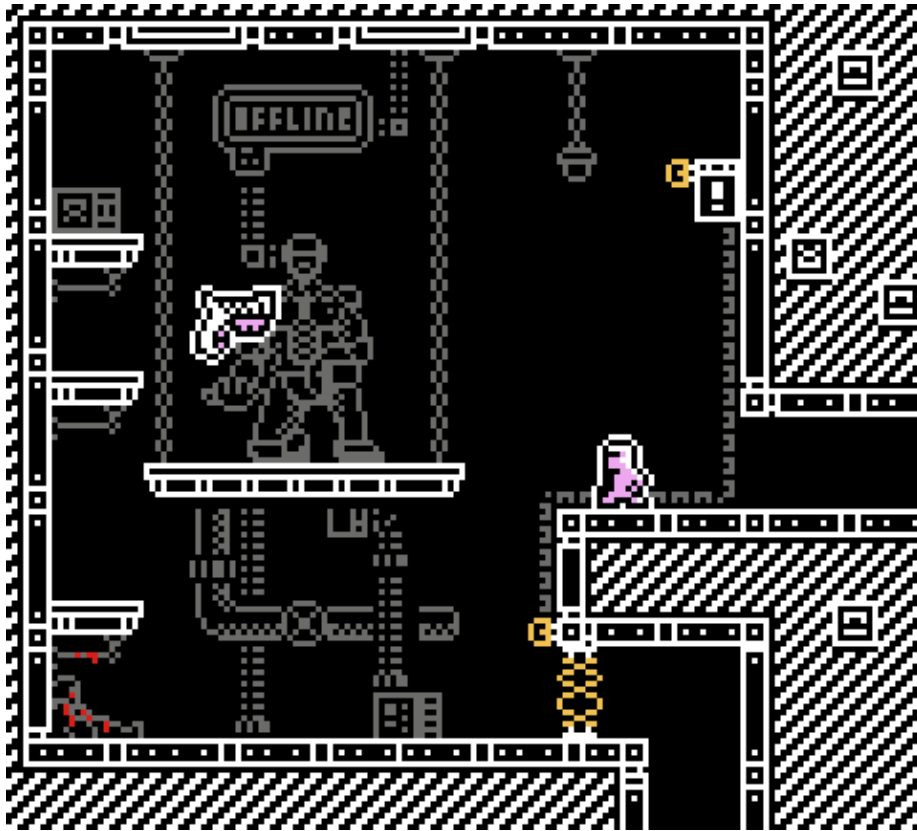
Ocean Base Stage in Mega Man Xtreme 2 (2001) In the Ocean Base stage from *Megaman Xtreme 2*, the highly saturated orange sand compliments the ‘blue’ sand that is submerged in water. By de-saturating the submerged sand as well, it makes it feel like the water is more viscous than the air around X. Both palette decisions have worked to accentuate the difference in player movement physics between X traveling through the air and water as a result.

1-Bit Palettes When referring to pixel art, palettes of this style use only 2 colors to convey visual information on screen at any one time. Due to its intrinsic simplicity, it’s a great way to explore color selection as an entry point, especially when learning about how specific colors can affect our own emotions – a useful tool when conveying mood in a narrative sense. The binary nature of this palette means you can utilize any two colors you like (although a typical example usually features black and white). I mention black and white palettes only because, thanks to the improved Game Boy Color screen, dark tones no longer smear when the view is in motion as with the DMG and Game Boy Pocket. This means the developer is free to finally use black as a background color without the fear of rendering their game unplayable on the older Game Boys. If you are thinking of using a 1-Bit palette, it’s important to note; It can be harder to direct the player’s attention towards a point of

interest when color and tonal contrasts are not an option. However, there are many other solutions to this problem: * Using movement/animation will draw the eye. * Increased detail/reduced detail over adjacent areas can also create contrast. * Incorporating lines into the background artwork or level design itself can act to direct the player's eye towards points of interest during game play.



The opening screen of Fix My Heart (2021) In Filipe Bianchin's *Fix My Heart*, the textured rock formation and the large white area surrounding the cave's door both work to draw the player's attention towards the right side of the screen, zeroing in on the way forward. Additionally, the single, unbroken horizontal platform creates a line from the player character, directly to the door. It is very clear what must be done. ## **Minimalist Palettes** By introducing a few more colors into the 1-Bit palette, we can easily draw the player's attention towards points of interest, dangerous obstacles or enemies, interactable objects etc. without getting bogged down in color management. Additionally, by using varied tones of the same hue, depth can be created within a scene while still maintaining decent readability.



Mandroid (Unreleased). In this scene from an unreleased proof of concept of mine, I have used white against black to create a foreground, an intermediate value (gray) to create background detail and some high contrast colors to highlight points of interest such as the door, the fuse box and the weapons upgrade. Minimalist palettes are quick to produce, keep the visuals clean and easily readable and focus the decision making towards game play above all else. It's another great way to dip your toes into the world of color without feeling like you may drown! ## **Retro Palettes** Before we let all the colors loose, another option is to self impose limitations. If you want a scene to be more colorful than a minimalist palette but are not ready to deal with the scope of the "anything goes" approach, then picking a palette from any number of the retro consoles is a great way to impose these limitations (and pay homage to great consoles from past decades too). It's an excellent way to explore how color can affect a scene without having to deal with an overabundance of choice.

A series of unlockable palette variations from *IN THE DARK* (2022). Shown here is a color scheme based off the Magnavox Odyssey (top left), Intellivision (top right), PICO-8 (bottom left) and the DMG-01 (Bottom right). A similar approach is to find ready made palettes online. I have found Lospec to be

extremely useful as it's a database for pixel art palettes – and therefore great for games. ## **All the Colors of the Rainbow** Going beyond the Minimalist and Retro palettes means the options are greatly expanded. For many, this is where choosing a cohesive, balanced palette that hits all the right notes is somewhat shrouded in mystery. But if we take things slowly, we will be able to navigate color selection for any project. The best place to start is by using one of the free tools available to us on the internet. Here is a great tool by Paletton or another by Adobe (and let's not forget Coolers!), if you want to start playing around immediately. These will do a lot of the heavy lifting for you, as different palette styles can be selected with ease. And by combining some of the readability principles discussed above, you will already be off to a good start. It's just a matter of transferring your chosen colors into the GB Studio palette creator tool and making sure they translate well to the 8×8 pixel tiles.

Let's take a look at an example. Below is a screenshot of a menu from *Dogz*, a



Game Boy Color game released in 1999.

The Petz Store Menu from Dogz (1999) There is nothing particularly wrong with the readability here. Yes, you can read all the information in the menu, but the various highly saturated colors in this palette are fighting one another for attention. Let's revamp this menu with a quick trip to one of the palette build-

ing tools I mentioned above, and attempt to bring some harmony to this screen...



The same menu but with a new split-complementary palette created in seconds using Adobe Color. ## Studying the Greats (and not-so-Greats)! Exploring the large catalogue of GBC games already available to us is a great source of inspiration (and for that matter, why not check out the NES titles, too) Studying the palettes of great games is well worth the time and effort but it doesn't hurt to analyze poorly crafted palettes as well – comparing what works to what doesn't work and understanding why will go a long way towards the creation of your own palettes.

The palette choices and readability on display are more a nightmare than a fairy tale. *But we can learn a lot from asking why! Shrek: Fairy Tale Freakdown*

(2001) What's more, studying the palettes of games rather than any old palette you find on the internet means you can see how that palette has been *applied* – after all, they have been crafted with game play in mind. This is the ultimate goal when it comes to deciding what colors to use in your game. Yes, it's great to have something that looks absolutely amazing but color is ultimately another tool in service of the game play. A great looking palette applied over some great looking pixel art can excite any retro gaming fan, but the bottom line is: it must also readily convey important game play information at the same time. The final thing I'd like to add is the most important principle of color palette creation, in my opinion. I've saved it to last in an attempt to have it burn into your memory. And that is: **## Keep it Simple!** The human brain tends to 'shut off' when overloaded with too much information. It's the same reason babies cry when they get over stimulated, or when we walk into a supermarket without a shopping list, we sometimes forget what we came in there for. In other words, too much information just translates to noise, and when we play a game that translates to disengagement or confusion in some way. Take a look at any bullet-hell and you will notice just how much attention is given to the color palette to make sure the player can easily read the huge amount of moving objects on screen. For this reason, the best thing for easily readable game play is to keep the on-screen palette focused. For those that shy away from the sometimes complex world of color, I hope this article has inflated your confidence enough to explore how color can enhance your own project. So why not tick that 'enable color mode' checkbox in GB Studio and start painting! In the next Chapter, we will explore the various ways a developer can maximize replayability in their game.

This article is part of a series: Development Workflow Gumpy Function

Gumpy Function

Independent Games Designer, Artist, Film Enthusiast and Full-time Dad (he/him). Check out my games here!

www.gumpyfunction.com/ **###** Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit **### Related**

Development Workflow – Chapter 9: Creating a Color Palette (Part 1)

A while back, I covered color theory in the context of game design. We explored how palettes can be applied to backgrounds and sprites in order to direct the

player's attention toward points of interest, liven up a scene with analogous colors, and improve readability overall. For the beginner, I suggested simplifying matters using 1-bit or minimal palettes, or by incorporating a pre-made palette into your own game scene. This article is for those who wish to go beyond the basics of color theory. We'll explore how to pinpoint a color you want, how to expand on that color to create a palette, and then organize that palette into a useful color ramp which can then be applied to any scene to make a cohesive whole. Before reading on, feel free to take this moment to read my chapter on color theory and refresh yourself on the various color theory terms and definitions that will be used in this article. # Picking A Color Pick a color, any color. It's simple, right? Well...of course it is! We just need to know what we are coloring and we're away. For the purposes of this article, I have drawn a simple tree using the default colors of GB Studio's .png template.

A tree, the art element we will be painting in this article. The first color that comes to mind when I think of a tree is green. But even something as simple as determining the colors of a tree's leaves can be a loaded question. What if you want to set your game in a season other than spring? Autumn or summer may have you wishing for a rich red or perhaps a dry yellow color. Or what if you want to provide a certain mood or tone that is specific to your game? Maybe you're making a horror game or some gritty dystopian adventure. Then you may be thinking a lush green color isn't the best solution. Maybe a dark blue or purple would be better? Or perhaps a green hue shifted toward orange or brown? The color you assign to any element *in-game* will depend on a number of factors. It's up to you to determine what colors will best suit whatever mood your game needs, but we will stick with good old green here. Something vibrant and inviting.

The first pick, a mid tone green. The most direct and simplistic way to extrapolate a palette from this color is to shift its *value* in both directions until we end up with a light tone, two varying mid tones, and a dark tone of the same *hue*.

The same green hue, but with four varying tones achieved by shifting the value parameter. What we have just made is a *color ramp* made of four colors, each of which we refer to as a *step*. Here it is applied to the .png file:

Our four-step color ramp applied to the tree GFX. # **Extrapolating Your Steps** Shifting the value has certainly given a sense of depth, but it's not the most thrilling collection of colors. In fact, it's not too dissimilar to the original .png we started with! Luckily, value is only one parameter at our disposal when coloring an element. To illustrate just how dynamic an image can get through palette selection alone, we're going to move out of the realm of pixels and into the world of fine art for a moment. Let's take a look at a painting by the impressionist John Singer Sargent to show you exactly what I mean.

Villa di Marlia: Lucca, John Singer Sargent 1910 I've selected this piece for its striking contrast between highlight and shadow. You can see that the grass is shifted towards varying yellow hues when in direct sunlight while areas of

grass in shadow have been heavily shifted in the opposite direction, towards blue. The stone masonry detail and red dirt path have been painted in the same way, mixing to create shades of purple. The end result is a vivid and exciting exploration of light and form.

A color ramp of colors that give the illusion of grass on paper from Villa di Marlia: Lucca. The highlight is a strong yellow but shifts towards a blue hue as the tone gets darker. But how does all this relate to selecting a palette for pixel art, I hear you say? Well, whether creating a digital pixel art tree or using analogue methods, such as with oils, watercolor, or any medium for that matter, hue, saturation and tone are all there to help create the illusion of form, or how light bounces off the surface or transmits through matter. So the same techniques and rules that an oil painter uses to create a palette before laying brush to canvas can be used just as effectively when a GB dev creates several palettes before painting any art element with some form to it. So with that, I'd like to share some rules that I use when not only creating a color ramp to paint a scene in a game but also when mixing my own oil paint palettes. # **The Rules of Color Shifting** If I had a mid tone red oil paint and wanted to create a highlight from this color, I would choose one of two colored paints to mix in. Firstly, I could choose yellow paint. It brightens the value of the red, and also shifts the hue towards a warm orange color. Or I could use a white paint. This would also increase the red's value but, this time, I would be shifting its hue towards a more cool color, in other words, blue. Same again for if I wanted to make the red color darker. I could pick a dark blue, mix it in, and watch the red paint turn to purple, slowly adding more until I get a darker purple and so on until I come to the desired colored paint. I would end up with some variation of a dark shade of cool purple. Or, I could add brown (which is a dark orange color if you want to pinpoint that on a color wheel) and I would end up with a darker warm red shifted towards yellow. In the context of digital art, shifting hues along a spectrum is akin to mixing in these yellows or whites, or these dark blues or browns into a base colored paint – only you are sliding a toggle on screen rather than physically mixing paint together, obviously. In any case, the same rules apply.

An example of the toggles you could move when shifting from a red to a lighter color in GBS' s Palettes view. In the above image I shifted the red mid tone color in several ways to arrive at a lighter yellow highlight color. Moving the brightness (value) toggle to the right lightens the color; that's simple enough. I also started fiddling with the *green* and *blue* toggles by shifting them to the right and left respectively. This in turn shifts the hue to the right and towards the yellow area of the hue spectrum. In this way, you can nudge any color you want in any direction you want to suit your needs!

Here are the rules in a handy table Let's return to our tree and redo the same four color ramp, this time utilizing the rules above:

The color ramp and tree GFX using colors shifted only with value (above) while the new color ramp shifts both value and hue (below) This new palette is a

bit more exciting than the original, and I could easily push the hues further apart to make an even more dynamic four-step palette. Experimentation is the name of the game here, try nudging colors in various directions, play with parameters, find out what works and what doesn't. Even though it seems like palette creation is some magical mystery for the beginner at times (and even for the intermediate), it really is just a matter of practice once you know the rules. Having said that, we aren't done yet! In Part 2 of this article, we will expand our four-color palette into a versatile *color ramp* with multiple colors, branching paths and color bridges. By the end, you will have the knowledge necessary to create a master palette that can color an entire game *and* ensure the game's palette is varied while still remaining cohesive as a whole.

This article is part of a series: Development Workflow Gumpy Function

Gumpy Function

Independent Games Designer, Artist, Film Enthusiast and Full-time Dad (he/him). Check out my games here!

www.gumpyfunction.com/ ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### *Related*

Development Workflow – Chapter 9: Creating a Color Palette (Part 2)

From Steps to Color Ramp

In Part 1 of this article, we learned how to create a step ramp from one color to another. Let's keep expanding our four steps into an expansive color ramp that incorporates multiple colors. This way, we can include some reds and browns and assign them to the tree's trunk and the flowers.

Our initial four-step palette and the Tree GFX we will continue painting Creating a *color ramp* can take a lot of work. Shifting colors around takes time and there is often a considerable amount of further nudging once you start testing out the ramp and realize some colors just aren't working well enough to support others. But it's all worth it! Because once you have a master palette in the form of an interconnected color ramp, it acts as an easy to use guide when creating any number of palettes within GB Studio for your own game. To start, let's pick one of the four colors, and start extrapolating from there. We could pick the dark green-blue color and shift towards a lighter red, passing through various

blues and purples, for example. Or we could start from the yellow and shift through to orange and then red and so on. When building your own, you can do whatever feels right for you but for now, let's start stepping to red from our yellow color.

Branching the color ramp out from the yellow color towards a darker red I have used the same rules described in Part 1, shifting the value to a darker tone as well as shifting the hue towards red as I go, *being careful not to create too small or large a step*. Now it's a simple matter of repeating the process, picking an existing light color and shifting it towards a new dark color or vice versa.

A more complete color ramp with a wide range of colors Now we are getting somewhere! This color ramp features a wide range of hues and all the steps are designed to shift gradually from one to the next. You can see that I could select any four linked steps here and insert them into a GBS palette to paint a tile or a sprite in-game with ease. Notice there are three branching ramps, one for blue colors and two gray ramps (one warm and one cool). Our yellow to red ramp is now a *color bridge*, and there is a large *color loop* making a circle across the ramp. We have just made a huge leap in complexity here, so I'll take this moment to break down what I did in a little more detail. # Using Image Editing Software Outside of GBS Personally, I find it easier to make a ramp outside of GBS using GIMP (or a similar image editing software). The *color picker tool* in an image editor is not only a relatively quick and painless way to nudge colors around but also visualize how far away each step is in terms of hue, value, or saturation. Here are some snapshots of some of the steps within our color ramp as an example:

The four blue steps in one of the branching ramps as seen in GIMP's color changer tool In the above example, I've shifted the dark purple towards a cool light blue by doing the following: * Increasing the value upwards, making the color more bright. * Decreasing the saturation slightly, making the color more "white". * Shifting the hue down away from red and towards blue. Take a moment to study the image below to see how the three points above can be translated visually using the color picker of your chosen image editing software:

The direction of movement through hue, value, and saturation as seen on GIMP's color changer tool As you can see, our four-color blue ramp is following the red directional arrows as indicated above. If we were to measure the difference between values from step to step, you will find that each is around the same. This will ensure each color steps up or down uniformly across the ramp. The same is true for the hue and saturation. In essence, you are looking to draw a straight line from the darkest color to the lightest color as you shift from light to dark and through the hue spectrum. How complex and interconnected a matrix of colors you want to create is entirely up to you. This color ramp may seem like quite a leap forward in complexity (and to be honest, it is), but the process is exactly the same as our initial four-step ramp from yellow to a dark green. The more you practice, the more comfortable you will become expanding a palette's complexity. Now it's time to test our colors by painting our tree GFX. In

this way, we can figure out if any of our selected colors aren't quite working to support the overall palette.

The tree GFX colored using some of the steps from our color ramp I've picked a few linked steps to color the trunk in brown and the flowers using some pink and orange colors. If we draw our attention to the pink flowers though, I think we could work on improving that strong pink color of the petals so that the flower isn't "jumping out" at us so much. I've purposefully made this pink all together too intense compared to the more muted highlight of the flower's top surface to illustrate an important point when creating a cohesive range of colors. And that is; manage your *saturation* or *the level of intensity of colors*. # Saturation Dealing with changes in saturation levels across a palette is perhaps the most difficult parameter to get right, at least in my opinion. Having said that, there is nothing wrong with changing the saturation of colors across a palette, in fact it's necessary to mute various colors and arrive at much needed gray scales or whiter variants of a color. At the same time, saturation contrasts can also be incredibly nuanced and difficult to get right! Let's return to the work of John Singer Sargent just to show you how effective creating a high saturation contrast between elements in a piece of art can be.

John Singer Sargent, A Dinner Table at Night, 1884 In *A Dinner Table at Night*, the vivid reds of the lamp shades and surrounding light are used to direct the viewer's attention to the female figure in the center of the painting. The rest of the colors are muted variations of blacks, grays, and yellows relative to the red. It is the contrast in color saturation that makes this piece such an effective use of color. In the context of game design, saturation contrasts can be used not only to create interesting static screens for cutscenes or menus, but also as a way to differentiate art elements to support the gameplay itself. For example, using muted colors can push background art further into the background, while using saturated colors will pull an art element into the foreground. As seen in the screenshot below, this can be used to improve readability and create a sense of receding dimension to a game's scene.

Samurai Kid, Biox (2001) As you can see, creating a strong saturation contrast within a scene can lead to great results. It can be a highly effective weapon in your game design arsenal, but if you are unsure about how to deal with saturation levels across a palette and want to keep things relatively simple as you learn, then I offer this advice to you: *Stick with a similar amount of saturation across a color ramp*. So if your game is designed to appeal to kids (or the kid in all of us), with the exception of shifting towards grays, keep the saturation of your color ramp relatively high. If you are going for a more relaxed or adult vibe, then you would ensure your color ramp features some degree of muted colors to support the game's premise. To reiterate, if you are ever unsure, as long as you don't start shifting the saturation's of mid and dark tones all over the place, your palette will work out just fine. With this in mind, it's time to nudge some colors in the palette. In particular we need to make sure the step between our flower highlight and the pink petal isn't too large of a tonal shift both in value

and saturation.

The pink color has been desaturated considerably and the purple has also been slightly desaturated. Now they blend more easily with the saturation level of the grass and don't "jump out" as much. We would then make the necessary edits to our master color ramp and continue tweaking and fine tuning it by applying colors to the scenes and art elements in our game, learning more about our palette and its strengths and weaknesses as we go. Now that we have improved our color palette ramp, we can create the necessary palettes in GBS and paint our scene.

A final touch-up with a .png prepared for coloring and the necessary palettes with a more uniform saturation created in GBS Picking a palette, creating a color ramp and fine tuning it can take time and quite a lot of patience. But it can also be a fun way to explore color as well as provide a base for you to come to when coloring your entire game in an effective and cohesive way. If you've never tried creating your own color ramp, I hope this article has helped to get you started. Yes, making a palette can seem like a perplexing problem but remember, start small, take your time, experiment and find what works for your own game. With a little practice, you will be creating not only aesthetically pleasing palettes but also palettes that can influence the very experience of the player in a substantial way, and your game will be all the better for it!

This article is part of a series: Development Workflow Gumpy Function

Gumpy Function

Independent Games Designer, Artist, Film Enthusiast and Full-time Dad (he/him). Check out my games here!

www.gumpyfunction.com/ ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### Related

Dissecting Classic Cutscenes From the Game Boy

I remember the day as clear as crystal, I had just walked into my science class in high school. The smell was pungent, nearly unbearable, I heard whispers over and over of the word "formaldehyde." The lights above were blinding and my knees were weak. My stomach tossed around like a ship at sea. At each table in the lab were six baby pigs, all dead. That day so many years ago was dissection day. Some felt excitement. All I felt was dread.

As morbid as dissection can seem, from a scientific standpoint, there are undoubtedly benefits. Today we will also be dissecting! Thankfully, not a cute baby animal, but instead some of our favorite Game Boy cutscenes. No smell, no bright lights. The purpose is to learn how some of the best game developers used the Game Boy's limitations to create special experiences on our favorite little handheld. In today's article I would like to show you a fun way to pick apart cutscenes from your favorite Game Boy games for learning purposes. ## Setup for Study Unfortunately, we don't always know exactly how the aforementioned cutscenes were made – however, there are ways to learn some of the clever tricks the developers employed. I hope thinking about these will help inspire you to know more of your options when working with GB Studio, and will result in helping you to create the best cutscenes for your game!

The main tool we will be using today is a fantastic emulator called mGBA. If you would like to conduct your own research on your favorite games, this emulator has a host of tools that can help you. You will also need a GB ROM of some kind, it can even be your own game.

Once you have opened mGBA and loaded your ROM the fun can begin. Navigate your cursor to the top left of the screen and select the “Audio/Video” tab. Near the bottom of this tab is the option “Video Layers.” From here you will see three options which can be toggled off. Two of these three options are the key to understanding the basic choices the developers made when creating their cutscenes.

These options will help us control what appears on the screen. Briefly here is what each of these options do: * Unselecting “Background” will remove the background of the cutscene. * Unselecting “Window” will remove any text, text boxes or elements related to overlays. We won't be using this feature for today's research, however it is still a useful feature to know. * Unselecting “Objects” will remove what GB Studio refers to as “Actors,” which would also remove projectiles. Let's demonstrate how toggling these options can help us better understand the thought process behind a famous Game Boy cutscene. Maybe you recognize this one:

No Changes

No Background

No Objects ## A Battle of Resources By taking the opening Pokémon cutscene apart this way we can determine what is happening here quickly. By removing the background we can understand that Gengar is drawn entirely with background tiles and Nidorino is the only rendered actor. One rationale the developers structured the cutscene this way is to prevent an overload of actors on screen at one time.

Since we understand the structuring better now, let's examine the techniques used so you can use them in your GB Studio project.

If Nidorino is the only actor, why does he appear behind Gengar? Aren't actors

rendered above the background? If that is your question then I am happy to introduce you to an incredible option every GB Studio dev should know about: “Display behind background layer.” To find this option you need to navigate to the Sprites tab on GB Studio. Then select the actor that you want to display behind the background layer. With a sprite tile selected, you can then tick the box “Display behind background layer.”

Literally just the tick of a box. How cool! Using this sprite technique and a simple “move actor” script, you can recreate a similar dynamic camera effect so that an actor and some background tiles can shift positions. With just a few tricks the developers created an unforgettable cutscene that captured the imagination of a whole generation of Game Boy players. What will you create by using this feature?

You may have noticed we didn’t talk much about how the developers were able to animate Gengar by using background tiles. The reason is because Pearacidic wrote a wonderful article about how you can use this technique in the article “Tile Swapping 2.0.” Please give that article a read because tile swapping is one of the most useful tools we have access to in GB Studio. ## A Perfect Storm of Techniques By examining the opening cutscene of Link’s Awakening we can immediately notice a few things that can help us push our own cutscenes to the max. When all the moving pieces are present, this cutscene looks absolutely packed with motion. Lightning striking left and right, rain coming down in buckets, and a ship tossing in a fast-moving ocean as the clouds swiftly pass. How can our favorite little handheld keep up? And even more importantly, how can we learn to use some of these tricks for our cutscenes?

No Changes

No Background

No Objects

When we examine what is happening here we get some insight into the developers’ sleight of hand. Without the background, we notice the ship doesn’t actually animate much more than a bobbing motion. The graphical artist cleverly carved a couple of wave-shaped cutouts on the bottom of the ship which creates an illusion of waves crashing on the bow at different heights when all together. Can you think of a way you could use transparency on your sprites to overlap with your background to create unique effects?

If you are counting lightning strikes and raindrops, you might be confused how so many sprites are being rendered on screen. Without the background, the illusion is a little more obvious, helping us learn there are really only two unique raindrops and three unique bolts of lightning. Reusing sprites in clever ways like flipping lightning strikes so they look like a different bolt is a technique we can use when trying to save resources in our creations. When you are creating the pixel art for your game, why not see how it reads if you flip it horizontally? You might be able to save time and unique sprite counts when crafting your cutscene

if you learn from these developers resourcefulness.

In the case of the rain, why not try this well loved GB Studio trick, which involves just one or two unique raindrops. Bryan Rollins succinctly describes the process here.

As for the background, we can see how the developers use motion to create immersion. The clouds are moving, the ocean is moving. Can you think of how using parallax scrolling could enhance your cutscene, or what if you could adjust the way the Game Boy draws to the screen to create effects similar to the waves in this scene? With the Scanline FX plugin you might be able to create an effect you never dreamed possible on the Game Boy.

When we are shifting scenes the screen has to cut to white or to black. If you are cutting repeatedly you might find the screen fades undermine the experience. However this subject of screen fades and cuts are not limited only to the Game Boy. If you study cinema, one thing you will hear over and over is a saying that good editing is invisible. A movie known for mastery of hiding screen fades is Alfred Hitchcock's "Rope." Notice just a taste in the clip below.

Did you catch the cut?

With that thought in mind, an underrated aspect of the Link's Awakening cutscene to me is the clever fade employed when lightning hits Link's boat. The screen fades just like a bolt of lightning that close to our faces would turn everything to white. How does dissecting the cutscene in this way help you think about the editing you have chosen to use in your cutscene? Can you think of an area where you can do some sneaky transitions?

That was quite a Trip...

We have been to battle and we have been to sea, now to our final destination of this research trip. Though Trip World is considered a "holy grail" of Game Boy collectors, the game's opening cutscene is not as well known as the last two examples I provided. However, there is more to learn from such a rich game. From this cutscene we can see a similar DNA of camera and background tricks used previously.

No Changes

No Background

No Objects

I felt it was necessary to research this cutscene though, as it displays a few more "need to know" tools at your disposal that we have not mentioned in this article, specifically how much palette-shifting and the use of sprites can benefit your cutscene when you have the resources to spare. Gumpy Function wrote in the "Creating a 'Reward' Scene" article the details of how to do just so.

In conclusion, regardless of how you feel about the limitations of the Game Boy, with some forethought and these tools, you can create the next memorable cutscene!

Editor's Note:

This article was submitted by Howdy Riceball. We'll be updating the article with a proper author bio once we get that built! howdy riceball

howdy riceball

Husband. Game Designer. Illustrator.

howdyriceball.itch.io/ ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### Related

How to Make a Responsive Action Platformer

Let's say that you have a great idea for the next big platformer title. You already have the levels floating around your head, the story is engaging and the gameplay is what gamers will rave about. You have planned: * Expert-level platforming level design * Intelligently crafted AI that really makes you think * A relatable main platforming protagonist that everyone will love The only thing you're missing is expertly-crafted, fine-tuned controls. We already have a great article about upgrading GB Studio's Platforming Engine, which will help you with coyote time, mid-air deceleration and jump buffering. All of these are great features to have, especially to ensure that players are having fun with your game and aren't required to have pixel-perfect jumping abilities. In this article, we'll add some additional tools to your arsenal beyond those to help players really connect with your game and ease them into the platforming requirements you've set for them. # Moves like Mega One of the most memorable things about the *Mega Man* series is the iconic jump physics. There's something to be said for a character who not only looks ridiculous while jumping, but still offers you tight controls to ensure you hit the mark each time you jump. With spikes, enemies, and all kinds of obstacles littering each level, the *Mega Man* games are iconically hard, but also respectable in how they presented your task. It was up to you to figure your way through the levels despite the odds. One thing that *Mega Man* had that GB Studio's platforming engine does *not* is the ability to turn around mid-air.

Mega Man 2 is known for its quick, twitchy controls In GB Studio, if you jump up and go to change to the other direction, you will find that you are stuck

with whatever direction you were facing before you made the jump. This is less than ideal, and could really frustrate when a player is trying to face an enemy to get some hits in. Luckily, this is an easy fix that doesn't require any engine modification, and it can be placed in your **[On Init]** event on any scene you intend to use it. # Fixing a Sticky Situation The reason I initially decided to add this to my game was due to a boss fight against a boss called the "Dreamweaver." The Dreamweaver is a huge spider hanging from the ceiling, and you must use webs to bounce up and hit him. When you bounce up from the webs, you are stuck in the direction you bounced from, which often leads Nemo to be facing the wrong direction:

Bouncing on a web and trying to hit the boss, but facing the wrong direction. Modifying the input setup to allow you to turn around after the jump makes a world of difference:

Bouncing on a web, Nemo is able to switch directions and hit the boss. Seems simple enough, right? So how is it achieved? Let's get into it! # Adding the Horizontal Flip Start by adding an **[Attach Script to Button]** event to your **[On Init]** script (make sure the checkbox for "override default button action" is unchecked). From there, select the Left and Right buttons to initiate this script, so whenever either one of them is pressed it will activate it. Next, add an **[If Joypad Input Held]** script to the "On Press" part of the 'Attach' script and select the Left input first. Within the Left's 'True' section, add a **[Set Actor Direction]**, click the drop down menu to select the Player, and select Left. This will allow the player to turn left if the jump was initiated facing right. To finish the script, add another **[If Joypad Input Held]**, but this time inside of the 'Else' section (click on the right-facing arrow next to 'Else' to open it). Go ahead and set this one to Right, and in the Right 'True' section, add another 'Set Actor Direction' with the Player selected, and set it to Right.

Adding our "Flip Script" to On Init on our scene With that all set up, GB Studio will now recognize that we pressed the opposite direction in the air, and switch accordingly! Another cool feature is that switching your direction retains a bit of the *X* or side-to-side velocity, so it also provides players some of what is commonly known as "Air Control," which is good for allowing additional, split-second decision-making in the air. # Attacking Any Which Way An unexpected, yet welcome feature is that you can now jump and use a projectile attack in whatever direction you are facing in the air. All you need to do is make sure you change your 'Direction' to Player in the 'Launch Projectile' settings.

Adding a Launch Projectile to our script. Make sure to select 'Player' as the Direction of your attack! Once this is set, it will also work for long-range attacks, as it will only shoot them in the direction the Player actor is facing. What if you want to have a melee attack flip as well? We'll cover that next. # Flipping Attacks like Crazy In most situations, when you're creating a melee attack in GB Studio, you use an Animation State to switch your player to look like they are attacking. If you don't and end up using a Launch Projectile script, you'll find that if you don't set the Life Time to almost instant, your attack just stays

lingering in the air from when you pressed the button. Pretty odd looking. The solution to this problem is a two-step fix. First off, using a custom Animation State as we mentioned, but combining it with an invisible Sprite Sheet like this one: An a Launch Projectile event, it will allow you to *fake* the attack and do all the dirty work behind the scenes. Now that you have that setup, you’ ll find that if you do your horizontal flip in the air after attacking, you will *never* hit the enemy. The reason why is that your Projectile attack will only work in the initial direction facing. To rectify this, we’ re going to add a little bit more scripting.

Adding a global variable so we can set up a ‘Flip Attack’ Let’ s break down what was added to this script. I am using the B Button as the attack button in this [**Attach Script to Button**] event. If you are not, change yours accordingly. Next I added a [**Variable Set To ‘True’**] event at the top of the script. I picked a Global Variable, and called it ‘Attack’ – or whatever you prefer. Next, change the player’ s animation state to an attack state, if you don’ t use a custom one to show that your character is attacking, then skip this step. Beyond that, you’ ll see the same Launch Projectile event that was added above. The [**Wait**] event is set to the same amount of the Life Time in the Launch Projectile event. This way, it changes the animation state back to Default at the same time. Finally, it is finished off with a [**Variable Set To ‘False’**] event, to reset the attack variable that was set at the beginning of the script. This basically gives the game a window of time that the attack takes place, so whenever it’ s finished it will turn off and no longer trigger a flip attack unexpectedly.

Add the launch projectile code to the horizontal flip to allow for a melee attack in the new direction. Finally, you’ ll need to add or copy the Launch Projectile event that was already made inside of an [**If Variable Is ‘True’**] event after the Set Actor Directions script in the script that was made earlier (both directions, this image only shows for Left). This sends out another attack when the direction is changed. Feel free to edit the Life Time — half is probably good — to reduce the attack timing. Otherwise it might be something players can exploit. Let’ s look at the difference. Here it is without the additional attack:

Nemo jumps and tries to flip the attack, but it doesn’ t work. And here’ s with the additional attack there. Much better!

Nemo jumps and flips after attacking, now it hits the enemy! # Wrapping Up Thanks to these simple fixes, you will gain additional control that you otherwise might not have even realized was missing. I guarantee it will make a tremendous amount of difference when playing, as nearly all platformers allow mid-air directional changes, and ultimately it *feels* better. Along with the extra attack on the horizontal flip, you’ ll have a really tight system that allows for even more control over how you interact with your platformer, and certainly, your players will thank you for it! I look forward to seeing what kind of great platforming titles you are able to create with these new tips. If you haven’ t yet, join the official GB Studio Discord and share what you’ ve done with our community! You can also play the demo for my game, Little Nemo: The

Slumberland Hero, on my itch page. Pearacidic

Pearacidic

Writer, Game Designer, Engine Bender, Retro Enthusiast. Co-creator of the In The Dark series and Grimace's Birthday. (He/Him)

pearacidicgames.com/ ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### *Related*

Making a Cutscene for the Game Boy in a Week

Can you think of your favorite cutscene from a Game Boy game? Maybe the cutscene that comes to mind involves a hero being struck by lightning on a raft, or two pocket-sized friends about to go in for a tackle like the defensive line in the last quarter of the Super Bowl.

Whatever your favorite cutscene may be, wouldn't you love to know the development workflow for such a cutscene? And wouldn't it be nice to add a cutscene to your own game that could inspire so much imagination?

Recently I was wishing for some insight into the thought process behind those iconic cutscenes since I planned to participate in the Game Boy Showdown 2025 Game Jam. This year the game jam was all about making a cutscene and moreover, the jam was only going to last one week. In this article I hope to share with you my experience participating in the game jam, as well as my thought process for brainstorming and developing a cutscene with the Game Boy's limitations. ## Is There Anything I Can Do Before the Jam Starts? Before the jam even started I did my best to identify any areas that would need the help of another. Not one second into this train of thought, I knew there was one major area I had absolutely no way of completing: Music & SFX. If left to my own devices the noise in my cutscene would sound more like I had dropped my Game Boy down a flight of stairs than it would sound like any sort of composition.

With that problem identified, a few weeks before the jam started I reached out to Beatscribe. I knew we worked well together because he had saved my poor music abilities in the past with the soundtrack to our game "a cat & his boy." Thankfully since I reached out early enough he was available to assist.

Since we can't start before the jam, all we can do is wait – right? Well, there are at least two other things I can do before we start. The first is quite simple: make an action plan to buy out some free time in my schedule to ensure I have

all my family, work, and other priorities reasonably settled. This allows me to theoretically have at least a couple hours free to work on the cutscene.

The second task I can do is research. I spent some time before the Jam started to analyze some standout Game Boy cutscenes from the past. In a future article, I will show you an example of how I conduct my research. Besides those items, all we can do is wait for the jam to begin and for the theme to be revealed, which will provide us with a prompt as a jumping off point. ## Ready... Set... Jam? Participating in a game jam is like what I imagine a runner feels when beginning a marathon. I did everything I could to prepare for this moment, there is no more stretching or training I can do at this time, the jam has begun!

To our surprise, there was no theme for the jam this year. We had total freedom as long as it was a cutscene playable on the Game Boy with a runtime of 60 seconds or less. So, what should we make? I think a lot of us Game Boy devs like restrictions; I mean we develop games for a piece of hardware that has many limitations. Some of those limitations are the best things about the console! Without a theme this changes everything. This led me to spending the first day of the jam a little lost by the infinite possibilities. However the next day brought new thoughts and I decided to just start drawing on some storyboard printouts I had on my desk in preparation for the jam. One of the pages I drew looked feasible in comparison to the amount of free time I anticipated to have. Here is what it looked like:

The initial storyboard sketch I sent this rough sketch to Beatscribe and after we exchanged some ideas and inspirations for the cutscene, we began to work on our tasks to make this animation an 8-bit reality! ## Inspirations, Innovations & Iterations From a development standpoint, I initially set out to create all the art quickly, which to me means in a relatively traditional pixel art style that you would expect to see on the Game Boy. My planning and workflow resembled more of a sculptor than a deliberate architect. So I shaped out large, blocky pixel shapes, and laid out my framework ready to chisel away like a freeform game of Picross.

I don't think this is what they mean when someone says "things are starting to take shape" From there I created a few assets. Although after some time I began to view them through a more critical eye. There was nothing inherently wrong with any of it at a glance, but for myself, coming from more of an arts & crafts background rather than a programming one, I wasn't happy with the artistic direction so far. Sometimes games emulate the classic look of the Game Boy, and that is good, but in this case I wanted the art to stand out. Game jams are a fantastic chance to try something bold since you will get immediate feedback if others like the style or not.

It's not bad, but it leaves no lasting impression At this point, I was starting to feel unmotivated to even participate in the jam since I hadn't really made any solid progress because of other important responsibilities. The clock was already ticking. We only had about 3 days left in the jam, but realistically, even less

since I had a lot going on at my day job. I took a breath and tried to create a problem list. My problems: * Limited time remaining. * Not happy with the art style. * Need a lot more art made. * Need to animate everything. When I am having problems creatively, I like to think about this really fantastic bit of advice: > “A good idea is something that does not solve just one single problem, but rather can solve multiple problems at once.” – Shigeru Miyamoto Is there any solution I can think of to solve two or even three of these problems at once? Let’s go back to the initial idea. The main character is a small toy knight made up of scraps of fabric, who uses a needle as a sword, has a thimble for a helmet, and a button as a shield. I got it! The inspiration was right there all along. The art needs to look like it is stitched together pieces of felt. That is an art style I love, but also it will be quicker to make this style of pixel art since it doesn’t require as much shading or dithering. The emphasis of the style is on stitchwork, not shadows and lighting. This solution will help tick the box on all of my remaining problems.

A close up of the more “stitched” pixel art style. I sent a build of the cutscene to Beatscribe, and one of the brilliant ideas he had was to use the Set Music Routine function to create a synergy between the programming, the art, and the music. This is one of those huge benefits of collaborating, I would have never used this amazing aspect of GB Studio if he didn’t recommend it. Since we agreed we wanted this synchronized feel, he would need a clearer idea of the timing of the animations for him to begin composing the music. So I rolled up my sleeves and made a few iterations of the cutscene in a rough form.

The animation test: with this we could begin to figure out the timing needed for the audio. With all of that in place, the rest was iteration. The last 10 percent of the project felt like more work than the first 90: adjusting timing, rearranging scenes and pushing pixels. Then before we knew it, time was up. We submitted our jam entry 2 hours early. What a relief! We made it in time. But even better, we made something we could be really proud of. Something artistically unique and original to things we are inspired by. Now all that was left was to see if anyone else felt the same way. The judging portion of the competition began! ## What was the result? To be honest, I wrote all of this before knowing what the result of the jam would be, because when I entered the jam it wasn’t about winning or losing but about growing as a game dev. I loved seeing what everyone else made! I can’t find it in myself to be competitive with other developers because what we make is all so different, there really is no comparison for me. I loved the other entries from this jam and had a huge smile watching them, knowing everyone worked so hard! The only thing I can compare this cutscene we made with is my previous work. So how does it compare? I would say it is a personal best so far, and I hope all the other people who participated feel the same way about their work. If you would like to see the final form of the cutscene along with the amazing music, please check it out along with all of the other Game Boy Showdown Game Jam entries: <https://itch.io/jam/game-boy-showdown-2025/entries> **Editor’s Note:**

This article was submitted by Howdy Riceball. We’ll be updating the article with

a proper author bio once we get that built! howdy riceball

howdy riceball

Husband. Game Designer. Illustrator.

howdyriceball.itch.io/ ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
 - Click to share on X (Opens in new window) X
 - Click to share on Facebook (Opens in new window) Facebook
 - Click to share on Mastodon (Opens in new window) Mastodon
 - Click to share on LinkedIn (Opens in new window) LinkedIn
 - Click to share on Reddit (Opens in new window) Reddit ### *Related*
-

Solving the 'A Button Paradox'

Many of you might be wondering what the *A Button Paradox* is. If you have come across this issue, it can be very frustrating. The *A Button Paradox* is when you add dialogue to a script attached to the *A* button, but on pressing *A* to close the dialogue box, your script is executed again, causing some undesired effects in your games. There are many reasons you might want to put dialogue in a script attached to *A* button. I personally use this for battle systems and some other things. So how do you overcome this issue? ## **The Method** This is the easiest method. You simply use a variable to check if the script is currently running and if it is,

▼ Attach Script To A Button ▼

◀	▲	▼	▶
A	B	Start	Select

A

☒ Override default button action

On Press

▼ If \$Local0 == 0 ▼

Local 0

Equal To
0

▼ \$Local0 = 1 ▼

Local 0

1

▼ Display "the dialogue at fault" ▼

the dialogue at fault

Add Avatar

▼ \$Local0 = 0 ▼

Local 0

0

+ Add Event

▶ Else

+ Add Event

you skip the script like so:

A local variable is used to bypass the infinite loop This took me hours to work out, mostly because I was overthinking it. Sometimes when you are working on a project for hours on end, you can sit there overthinking a task that has quite a simple solution. Breaks do help and if I had done so earlier, I might not have had such a hard time working this out! Veund

Veund

Game Boy Enthusiast, Hardware Fanatic, Mad Man. (he/him) ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
 - Click to share on X (Opens in new window) X
 - Click to share on Facebook (Opens in new window) Facebook
 - Click to share on Mastodon (Opens in new window) Mastodon
 - Click to share on LinkedIn (Opens in new window) LinkedIn
 - Click to share on Reddit (Opens in new window) Reddit ### *Related*
-

Ten Lessons I learned During My Latest Project with GB Studio

After three months of trying to find time, I finally managed to release the demo of my game just in time for the GBCompo25 game jam. Although I' ve been making small games with GB Studio for over a year, Fun Video Store is my first truly serious project where I had to plan things properly to see it through. Being a solo dev is liberating because you' re your own boss, but it' s also risky: one misstep in planning can make you want to throw in the towel halfway through. That' s why I wanted to share some of the things I' ve learned over the past few months, both from this project and a few smaller ones I made before. I realized just how fascinating the world of creating games for the Game Boy really is. ## Don' t aim too high Defining your scope is crucial. It' s what ruins 90% of projects that developers start. In the first two weeks, I realized there was no way I could finish the whole game in 90 days, so I set milestones to at least have a playable demo. Your enemies are perfectionism and unnecessary features. You don' t need a Trello or Notion board packed with sections and flowcharts, so an Excel sheet with a few tabs can keep you perfectly organized. Set realistic goals with deadlines, advance a little bit each day, and don' t overdo it. ## Plugins are handy, but heavy Some plugins let you do things that GB Studio can' t handle by default, which is awesome, but they can slow your game down. You' ll notice it halfway through development when that Metroidvania you' ve been building for three months starts lagging as soon as two enemies appear on screen. Simple problems often need simple solutions, and if you need an extra boost, learning a few tricks with GBVM is always an option, as long as you have some basic coding knowledge. ## Write down your ideas Keep a notebook and pen with you. Sit in a park and brainstorm. Sketch ideas, write a 5-page synopsis, even if it' s just a clone of Blockout with talking furries chatting between levels. Ideas in your head tend to mutate and lose sense if you don' t capture them.

Writing them down lets you discard the noise, and focus on the genuinely brilliant ones. Sure, you might solve a problem in four different ways while taking a shower, but forget most of them the moment you dry your back. Take advantage of those bursts of creativity. ## Use version control Whether it's a GitHub repo or just 20 copies of your GB Studio project folder, make backups every time you make a big change. Your 2013 i3 laptop won't last forever. The same goes for visual assets: that animation you thought was useless could save you weeks later. Everything is reusable, so set aside some space on your local drive for discarded stuff, even if it doesn't seem related to your current project at first. ## Game jams are a great playground The best way to build confidence is to prove you can finish what you start. Your first project doesn't need to be a massive JRPG that could have ruled 1995 (trust me, I tried that and went insane). To test your skills, it's easier to find a free weekend than a whole year. Check out jams on itch.io or join local Global Game Jam events. They're perfect for learning to work under pressure and turning it into a super fun creative marathon, ultimately proving that you CAN do it! ## Events order matters If your scenes are complex, keep things organized when distributing events across actors and scripts. Even though execution is sequential, some "on init" effects can trigger earlier than you'd like, causing unnecessary delays or weird glitches. Two options: control the fade-in manually so the scene only shows when everything is ready, or centralize the whole sequence in one place. Your future self will thank you. ## Customize everything Don't ask why, but many players skip games that show GB Studio's default UI in screenshots. It doesn't take much to swap in a custom font or draw your own dialogue box. The beauty of GB Studio is making something that looks like it could have been natively developed for the Game Boy, something that makes you go "wow, this could have been released in 1992!" A bit of polish goes a long way, and interest in your project will grow enormously. ## Music is 50% of the experience (At least according to George Lucas). Chiptune music isn't that hard to make, especially with GB Studio's built-in editor. And if composing isn't your thing, there are plenty of sound artists who'd love to collaborate or share their tracks. In my case, defining the two main songs completely changed the vibe of my game. Oh, and remember: if your game relies heavily on sound effects, keep one channel free so you don't mute the music track. There's nothing more charming than hearing sweet melodies come out of a tiny 0.5W speaker. ## Remember, there's a Color-Only mode Making games for the old DMG Game Boy is charming thanks to its cute 4-color palette and strict limitations. But don't forget GB Studio also supports Game Boy Color-exclusive mode: faster CPU, more VRAM (so more space for unique tiles per scene!), and access to the tile priority brush. In my case, it was essential because of the number of elements in some scenes and the use of a condensed font. Let's be honest: most people will play your game on a browser or a Chinese handheld, not on a perfectly modded DMG with an IPS screen. That extra power is worth it. ## Your life comes first Most of us dive into GB Studio because it's amazing for making games fast, but even as a hobby, it can be a rabbit hole. Redditors know the sleepless nights you can spend trying to get an overlay to line up. Keep in mind: it's 2025, you're

making a Game Boy game that maybe 200 people will ever play. Perspective matters. Only if true passion drives you, and it doesn't negatively affect other aspects of your life, go for it! Raúl Rosso

Raúl Rosso

Communication specialist. Over a decade in the mobile app distribution industry. Obsessed with game design, creativity, and homebrew. I make video games just for fun. (he/him)
metaludica.com

elrosso.itch.io/ ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### *Related*

Top Five Mistakes for First Time GB Studio Users

Every piece of software has a learning curve. Some may be very steep, forcing you to absorb a lot before you can really dive into using it. Some, like GB Studio, have a fairly shallow curve, allowing a lot of people to dive right in and start creating. However, even with a user-friendly interface, there are common mistakes that befall beginners when using GB Studio. Here's some of those mistakes to avoid if you're just starting out. Checking these out beforehand will hopefully make your experience with the software more enjoyable. # Not Reading The Docs GB Studio has a really awesome interface. Almost all the information about the software is provided to the user through the program itself with tool tips and warnings, but there's still a lot of important aspects that need to be known about it before getting started with a project. It can be really tempting to just hit the ground running and start building your game, but chances are you will hit a snag when it comes to a specific detail, some are listed below. Software Documentation in general can be a slog to read through sometimes, but GB Studio has one of the best setup docs out there, broken up into the different areas of development quite well. I read through the entire documentation before I started playing with the software and it was quite advantageous. It's a great idea to give the docs at least a brief overview before starting to build a game. The documentation for GB Studio's 2.0 Beta did not see an official release, and it may not happen due to development moving to 3.0 now, but you can still find an incomplete set of docs that goes over many of the new tools and features (like what the background size limits are) here: <https://develop.gbstudio.dev/docs/> – we've listed this at the top of our Resources page for easy reference in the future. *Note: Viewing the docs is much better in a desktop browser thanks to*

the navigation sidebar. # Not Running The Latest Beta This one could be a bit controversial. Beta builds are supposed to be software that's in development and not fully released. They can have bugs and potentially leave you dissatisfied with a product, but GB Studio is different. GB Studio's 2.0 Beta is actually quite stable and has lots of quality of life features and bug fixes that will make development easier compared to the latest official release of 1.2.1. Things like increased background size, a redesigned layout/UI, better music implementation (less slow down), and more triggers are super advantageous to have at your disposal even if you ignore all the color enhancements. What isn't so obvious is that while GB Studio's official Itch page hosts what is known as the 2.0 Beta 4 build, there is another more recent build, Beta 5, available through GitHub. Beta 5 has a series of fixes that resolve many issues people were having with behaviour in game and introduces a new UI. Having Scenes, Custom Events and Variables listed in a sidebar is really handy to have when building your game, and little tweaks like Text previews are very nice. You can get the latest Beta for 2.0 here: <https://github.com/chrismltby/gb-studio/tree/v2beta#beta-builds> – This will likely be the last build of 2.0 as development is now active on 3.0.

If you don't see "beta5" in the "About" window, you're missing out. # Not Using The Correct Colors This personally tripped me up the first time I tried using GB Studio. The Game Boy uses 2-Bit Per Pixel color depth, which translates to 4 colors: White, Light Green, Dark Green and Black (technically they are all shades of Green, but we'll use White and Black for simplicity). Backgrounds in any Game Boy game are able to use all four of these colors, but Sprites can only use 3 if they are to have transparency, and pretty much every sprite will have transparency. When developing with traditional Game Boy development tools, you can determine which of the 4 colors gets used to represent transparency, but with GB Studio you can't. Sprites cannot show "Dark Green", and instead use a "Bright Green" to determine what parts of the graphic are transparent (think Green Screens). If you don't take this into account when you start out, you may include colors in your sprite sheets that won't render as you expect when you compile your game. In 3.0, the new meta-sprite tools will allow for 4 color sprites and you'll be able to split sprites apart, but until then this continues to be a common problem seen for first time developers. Luckily, the docs have premade palettes to help you out. Check out these specific pages for more info: <https://www.gbstudio.dev/docs/sprites/> and <https://www.gbstudio.dev/docs/backgrounds/>

I literally screenshot these from the docs – have you read them yet? # Mixing Local and Global Variables This one can really creep up on you if you're not aware of how variables work in GB Studio. GB Studio offers 512 Global Variables that can be accessed from anywhere in game. There are also 8 unique Local Variables that are available for every Scene and every Actor within your game in GB Studio 2.0. These Local Variables are great for tracking things like opened chests and dialogue with characters, but they are not accessible from any scripts or events outside of those specific Actors or Scenes. The easiest way to avoid confusion is to commit to only using Global Variables, which for a while was

the only option available anyways. If you're having issues with one of your scripts, make sure to check which variables you are comparing, chances are you may have mixed up one of these by accident. Also, we wrote a Basics article on Variables, so that should help. # Getting Too Ambitious Being introduced to GB Studio can be extremely freeing. Perhaps you have had this amazing idea for a game, and you want to try and implement every aspect of it. The problem is that unless you are an experienced game developer, it can be difficult to gauge exactly what can and can't be done. It's a really good idea to keep things super simple for your first game, or possibly just trying to build parts of it to test and see if it's possible. We wrote a whole article about what to consider for your first game, and that mentality sticks true.

Zen and the Art of Retro Level Design in “Kudzu”

Christ Totten is a game developer and professor trained in architecture who released An Architectural Approach to Level Design in 2019. His latest work, Kudzu, is a top down adventure game built in GB Studio that blends adventure and exploration elements from Zelda with non-linear progression like that found in Metroidvania style games. He wrote an in-depth article about the game's development process, and has allowed us to re-publish a small excerpt from it that focuses on the level design.

III. Music & Sound

- FAQs for GB Studio Music
- GB Studio Music Lesson 1: Terminology and Basics of the Game Boy Sound Chip
- GB Studio Music Lesson 2: Noise Channel / Drums
- GB Studio Music Lesson 3: Duty Channels
- GB Studio Music Lesson 4: Wave Channel
- GB Studio Music Lesson 5: Effects
- GB Studio Music Lesson 6: Advanced Drums
- GB Studio Music Lesson 7: Special Tricks Part 1
- GB Studio Music Lesson 7: Special Tricks Part 2
- GB Studio Music Lesson 8: Common Mistakes
- Getting Started in hUGETracker
- HugeTracker Part 6: Subpatterns
- Making a Theremin Sound in GB Studio
- Sound Effects and Music in GB Studio 3.1
- VGM Sound Effects Part 2
- VGM Sound Effects Part 3
- VGM Sound Effects Part I

- hUGETracker Effects Part 2: Tempo, Pattern Break and Pattern Jump and Stereo Panning
- hUGETracker Lesson 2 : Noise Channel
- hUGETracker Lesson 3: Duty Channels
- hUGETracker Lesson 4: Wave Channel
- hUGETracker Lesson 5: Effects - Portamentos, Volume Control and Note Cuts

FAQs for GB Studio Music

Creating music for GB Studio can be difficult. For musicians coming from modern Digital Audio Workstations, using trackers can be a difficult switch of mindset. Even for musicians familiar with trackers, there are a lot of restrictions in using GBT Player that don't mesh well with the traditional .mod composition process. This FAQ covers 7 questions that I've received the most from composers looking to learn more about GB Studio and GBT Player. If you're just looking for easier ways of getting music into your games, consider browsing the GB Studio Community Assets which contains free music that works in GB Studio. Otherwise, if you're still here and ready to learn more composing for GB Studio, then buckle up! This FAQ will cover the different types of files you can use, how to change song speed and time signature, and troubleshooting your compositions.

Can I use wav/mp3 files? This is the most asked question I get about music in GB Studio, and I'm sorry to say, the answer is no. Now, it is *possible* to play samples in a Game Boy game, but doing so is very intensive on the hardware. A great video on this topic is RetroGameMechanicsExplained's video on Pikachu's voice. Trying to play audio samples and keep the gameplay at a stable framerate requires a fair amount of control over the game's code to ensure the game doesn't crash or experience major slowdowns. We may eventually see sample playback come to GB Studio, but it will likely come with some restrictions, such as being reserved for cutscenes or for short 2-3 second clips.

What music files does GB Studio accept? GBT-ready Mod files (*they end with .mod*) are the only file types that GB Studio 1.0 and 2.0 Beta will currently accept. These are files that were composed specifically for use in GB Studio. Luckily, with some editing work, you can convert other file types to a GBT-ready file. These file types include: * Standard Mod (.mod) * Extended Mod (.xm) * MIDI (.midi) * FastTracker (.ft – when exported as text) * FamiStudio (.fms – when exported as FastTracker text) To convert FastTracker text, use ft2mod, a webtool by James Park: <http://ft2mod.jamespark.ninja/> To convert .midi files, use midi2mod, another webtool by James Park: <http://midi2mod.jamespark.ninja> There's also ajbookr's youtube video on manually converting midi files to mod for use in GB Studio using OpenMPT: https://youtu.be/Qz7z7yWn_5w Unfortunately, converting random mod or xm files to a GBT-Ready file has varying success depending on how the original file was made. Some instruments can be tuned differently, so the note names will be transposed incorrectly. This is a process that's more difficult to do and will be covered in a future article.

How do I change how fast my song plays? To change your song speed, pick any channel,

go to the effects column of that channel, and add **[F05]** to change your song speed to 5. This number **05** represents how many frames each row should take up. Speed 05 uses 5 frames for each row. To make your song go faster, enter a lower number like **[F01]**. To make your song go slower, enter a higher number like **[F09]**. # Can I use triplets/quintuplets? Yes! For the musicians reading, you have to consider which subdivisions you want to use in your song. Feeling like writing triplets? Consider writing your melody with 2 empty rows between each note, so you have 3 rows-per-beat. Here's what that would look like: This example song is primed for triplets, just fill in the empty spaces to create triplets instead of eighth notes. If you want to use quintuplets instead of triplets, put 4 empty rows between each note, so your song adds up to 5 rows-per-beat. When you run out of space, use a **[Dxx]** effect to jump to the next pattern. Just be careful to replace it with **[B00]** if you're on the final pattern! If you've composed a song for GB Studio before, you may have already used 2 rows per beat without realizing it. Here's some examples of how to get different subdivisions when composing. |||| **

GB Studio Music Lesson 2: Noise Channel / Drums

This is Part 2 for the GB Studio Tracker series. Part 1 can be found here if you haven't read it yet. In this lesson, we are going to make drums! The best place to start when making a song is with simple placeholder drums. This helps you make sure you're not imagining the tempo being different than it actually is. We will start by making a metronome to help hold your song together. Later on we will go back and make more realistic drums. But, one step at a time. # **Making a New Song** In GB Studio's Music Editor, Click the "+" button on the top left next to "Songs" in the sidebar and you'll see *song_template_1.uge* appear on the side (it may be 2 or 3, which doesn't matter). Click the new *song_template_x.uge* that appears. You can go into */assets/music* and change the name of this to something you want it to be: *my_demo.uge* or whatever. ## **Making Your First Instrument** If you're using GB Studio 3.0 or higher, you'll find that some basic instruments already exist in the template. We are going to overwrite them in order to learn what everything does. Don't worry about erasing them, you'll get them back whenever you make a new song in the future. We are going to make a basic hi-hat instrument. If you don't have any music background, the hi-hat is a small cymbal next to the snare drum that can be open and closed to change the way it sounds. A hi-hat hits softer than a snare drum and has a quick decay. It's a short, sharp sound. But first, it'd be good to learn what the noise channel actually does by default before we try to control it. Click the triangle next to the **Noise** heading on the sidebar on the left and then click on **01: Closed Hi-Hat**. Change the settings of the Noise 1 instrument to match this:

A not very exciting instrument, but good for demonstration If you hit the test button you won't hear much. Just a little rumble, not very appealing. That's because the noise channel doesn't make much noise at lower octaves. Don't

worry about it now, you’ ll get to hear it in a moment. Make sure the editor is in Tracker mode and set the dropdowns on the top to **Instrument 1** and **Octave 6** (see image below for reference). Click on the first slot in the Noise channel (set of three dots, “...”) and push **A** on your keyboard – you should hear a note play and “C-7” will appear in there. Notice how every 8 rows the shading of the cell changes? We can imagine these as beats in a song, and each row divides these beats into 8th notes. In GB Studio’ s tracker, there are 64 rows in a pattern, so you can consider an entire pattern like an 8 beat measure or “chunk” of a song. You can also think of them as 2 sets of 4 beats if that’ s easier. To flesh out the beat or pulse of the song, put the same C7 note on every colored line (see the image below).

A very simple beat, with a note every 8 rows You’ ll notice the note name (**C-7**) shows what is going to be triggered when the tracker hits that line. Next to it is **01**, which is the instrument number, corresponding to how they’ re listed out in the sidebar on the left. Now we can finally press play! Ouch! The static blasting out of the speakers is not really what we want. We’ re going to apply an envelope, which will shape how the sound changes in volume over time. Change the Initial Volume to **8** and Sweep Change to **-7**:

Setting the envelope parameters Hit play once again. Now it sounds like hi-hats. We basically told the system, “every time you reach a note on the noise channel pattern, turn the volume up real fast and then back down right away” . This is what the line graph is representing for us under those settings. Adjust Sweep Change and play it again to hear the difference. You can also click the length box and then specify how many ticks should go by before you cut off the sound. We won’ t apply this here, but will in later lessons. ## **Making a Snare Drum** Before we move on, let’ s make a snare drum to show off a little bit more of the noise channel settings. Go to Noise Instrument 2 by clicking on it on the left. Adjust the settings as shown here. What’ s the new section you see at the bottom that looks like a bar chart? This will make tiny increments in the note’ s pitch every tick (a very short amount of time in the song). In the previous example, since the graph was flat there was no change in the note. In this setting, the pitch will increase quickly. Notice we also put the Initial Volume higher. At the top of the tracker, set it to **Octave 4** and add an (**F-5**) note (**H** on your keyboard) to lines **16** and **48** in the pattern. Press play and you should hear something resembling a simple beat now. Great job! You are underway in making your first song! We will come back to drums in the effects section and do some more tricks. But for now, save what you have and get ready to continue. Now we’ re ready to actually make some musical melodies play. The next lesson will cover the Duty channel. Download The GB Studio Demo for Lesson 2

This article is part of a series: GB Studio Music Beatscribe

Beatscribe

Beatscribe has composed and produced soundtracks for games on Nintendo DS,

Nintendo Switch, PS4, Xbox1, Sega Genesis and numerous iOS and Android games for over a decade, but his true passion is creating epic moods on ancient hardware like the Game Boy and NES. ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
 - Click to share on X (Opens in new window) X
 - Click to share on Facebook (Opens in new window) Facebook
 - Click to share on Mastodon (Opens in new window) Mastodon
 - Click to share on LinkedIn (Opens in new window) LinkedIn
 - Click to share on Reddit (Opens in new window) Reddit ### Related
-

GB Studio Music Lesson 3: Duty Channels

This article is a part 3 of a series, if you haven't already check out Parts 1 and Part 2 first. Now we've come to making actual music notes with the two Duty channels in GB Studio's tracker. The Duty channels are very versatile and usually make up the rhythm or melodies of a song. Things will start to get a little more complex here, but you'll see how new things build on what we've already learned. **Making a Duty Instrument** The good news is that making a Duty instrument works almost exactly like making a Noise instrument as we did in the previous lesson. There are a few small differences, which we will highlight when we make the lead instrument for our sample song. The instruments we make for Duty channels can play in both channels at the same time or in conjunction with different Duty instruments. Much like you did with the Noise instrument, select the first Duty instrument by double clicking it on the left. You will see the following settings for the instrument. Your settings may not fully match this. That's ok. We will adjust it in a minute. First, let's learn what's new compared to the Noise channel. The first few settings are just like the Noise channel. You can set an optional length. Initial Volume and Sweep Change let you design your volume envelope, just like you did when you made a high hat. However, there are a few new settings here. Sweep Time and Sweep Shift allow you to produce a note that rises or falls in pitch over a certain period of time. Think of a bomb dropping in a cartoon, that kind of sound. Only the Duty 1 channel has this ability. If you use an instrument with a sweep on Duty 2, it simply won't bend in pitch. We will use this setting more later on. The last new thing is the Duty setting. This changes the "thickness" of the note. Try changing it and hitting Test Instrument. You will hear it sounds like a thin, reedy instrument when it's on 12.5%, like a synthesizer at 25% and 75%, and at 50% it sounds more like a flute. For now let's set it to 25% and set up the envelope as shown above. Now we can lay down our notes just like we did for the high hat. This will be the first time you have to build a complex melody in the tracker. So let's learn a little bit about how to put down actual music notes. Your computer's keyboard basically becomes a musical keyboard in a tracker. Here's how the notes are laid out. Depending on the selected octave, the notes will be different pitches, highest on the top level and lowest on the

bottom row. Time to enter notes. It will not be an easy first experience. Laying down a melody is probably the most tedious part of working with a tracker. Use the keyboard to find the right notes as shown here. Here's some other shortcuts that might help: * Del – Erase the note under the cursor. * Backspace – Erase the note under the cursor and shift notes under it up one. * Cut/Copy/Paste – You can also cut and paste just like you would in an Excel spreadsheet. The keys differ if you are on a Mac or a PC, but you can google it if you don't know how to copy/paste. Add the notes as shown here. Unfortunately it takes 3 screenshots to show them all. Now press play. We are still a long way from the finished demo song we heard in lesson 1. It is slower, it lacks some elements, but the song is starting to take shape. Since we did the drums first we have a clear structure and beat. Try fiddling with Duty 1 instrument settings to see how it impacts the sound of the song. For now, we will leave Duty 2 alone. We will come back to it once we've learned some more tricks in a future lesson. Now our song is still lacking one major element, the bass! We will learn how to make some beefy bass with the wave channel next! Download Demo UGE File for Lesson 3

This article is part of a series: GB Studio Music Beatscribe

Beatscribe

Beatscribe has composed and produced soundtracks for games on Nintendo DS, Nintendo Switch, PS4, Xbox1, Sega Genesis and numerous iOS and Android games for over a decade, but his true passion is creating epic moods on ancient hardware like the Game Boy and NES. ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### *Related*

GB Studio Music Lesson 4: Wave Channel

The wave channel is the most distinctive and versatile part of the Game Boy's sound chip. It can go an octave lower than the duty channels, and can be configured to produce a wide variety of sounds such as bass notes, kick drums, crunchy guitars, and even recorded wav samples. It can also make very unique higher octave notes that sound like flutes or bells. In this lesson we are going to talk about the basics of making a bass line for our song. The bass is the lower end of a song and in non-chip music it usually comes from a bass guitar, but can also be from keyboards or synths. The Game Boy chip's bass is crunchy and has lots of character. You can configure it to be quieter and smoother, but its lower octaves will also get quieter. So a strong bass line is usually the job of the

wave channel. Effects are very important in controlling the wave channel. We are going to keep it simple for today and look at setting up a wave instrument and putting it into our sample song. We will come back later once we learn about effects to learn how to do more fancy tricks like bass kick drums. # **Making a Wave Instrument** Much like the noise and duty instruments, there are a wide variety of premade wave instruments. We are going to make one from scratch to learn how it all works. Click `wave1` in the side panel. You'll see that the instrument panel has similar settings to the duty channel. You can't make an envelope, but you can set a length and choose one of four volume levels. Make sure ours is at 100% volume and has no length (meaning it plays indefinitely). Unique to the wave channel is the wave option. Below the volume area (where you'd see sweep settings for duty), you'll see a grid set up to modify the wave itself. The waves are not tied to instruments, you can click the waveform drop down and pick a different one. For now let's leave it at **wave 0**. Click the **Test Instrument** button to hear how it sounds now. Now click around on the grid, make some mountains and valleys, just for fun. Now press "Test Instrument" again. You'll be able to hear the difference as you edit and test the wave. For now, let's take a look at a few things to learn how it works. Draw this next shape or select waveform 11 from the dropdown menu. Remind you of another Nintendo system? Now it sounds like the NES triangle channel. It makes for a soft bass (too soft for GB hardware honestly) or a flute-like sound at higher octaves. Now, change it to the following form: This shape creates a common synth waveform called a saw tooth. As you can hear, it has a slightly harsher tone compared to the "smooth" triangle. It makes a great bass for a high-energy song. Another softer waveform is the sine wave: The sine wave isn't super exciting as a bass instrument, but if you put it on higher octaves, you can get some softer piano or bell-like tones. You can see how powerful the wave channel can be. For our sample song we are going to use the waveform 2 that comes with GB Studio, or you can design your own! You can edit the wave after putting down notes and the sound will change automatically. # **Composing Notes** We're going to lay down notes just like we did for duty channels. Lower the octave down to 3. Select the Wave 1 instrument and enter the notes as shown, only three note for now. Put a **C4** on row 1 by pressing `Q` on your keyboard. Next, on line 32 a **D#4** by pressing `3`, then lastly on line 48, press `R` for an **F4**. Now it's got some good bass! When we play the song back we can see it's starting to sound like the final piece. Right now the notes keep playing unlike in the final, but we will come back and learn what to do about that after we understand how to use effects, which is what we cover in our next article! Download GBS Music Lesson 4 .uge File

This article is part of a series: GB Studio Music Beatscribe

Beatscribe

Beatscribe has composed and produced soundtracks for games on Nintendo DS, Nintendo Switch, PS4, Xbox1, Sega Genesis and numerous iOS and Android games for over a decade, but his true passion is creating epic moods on ancient

hardware like the Game Boy and NES. ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
 - Click to share on X (Opens in new window) X
 - Click to share on Facebook (Opens in new window) Facebook
 - Click to share on Mastodon (Opens in new window) Mastodon
 - Click to share on LinkedIn (Opens in new window) LinkedIn
 - Click to share on Reddit (Opens in new window) Reddit ### *Related*
-

GB Studio Music Lesson 5: Effects

By now you’ ve learned enough to actually make something that sounds like a real song. But you might notice we are still lacking a lot of polish. Little things like notes that fade or echoes that sound distant. Even one big element, our four to the floor bass kick drums are not present at all! All these elements require using effects. Effects let you do stuff like pan the stereo of one channel or make a note bend up or down. On the wave channel the effects even let you control the volume. They also control song level events like speeding up or slowing down the speed (tempo) of the song, or jumping to a certain point in their song to make it loop. We will look at some common useful ones here with a simple demo file. The effects do not work the same on every channel, so we will talk about those differences too. There is a lot to go over so we will look at all of this over the course of two articles, culminating with applying what we learned to our demo song. # Hex Values We are going to avoid getting too technical in this article, but it is important to understand hexadecimal values when using effects. Hexadecimal is explained here but basically it’ s just 0-9 and then A-F to represent up to 16 numbers in one character. It is also used to compose groups of true and false values with binary math; this is what you will see with the stereo effect as you read on. Fortunately, you don’ t have to fully grasp all of this or do the math by hand, as the GbStudio UI will help you with it. # How to Place an Effect Effects are laid down just like notes, but one column over in GB Studio tracker. Make a new empty song. Press the + to do this and find the song template on the side. Put down a note in duty 1. Doesn’ t matter what note. Put down some default hi-hats and wave too. Align the cursor as shown in photo: Type an 8 on your keyboard. You’ re expected to enter two hex values next. What are they? Well, it’ s different for each effect. You can dig into the manual or you can use the handy UI which appears on the side panel when you enter an effect code. It turns out 8 is for setting the stereo panning of each channel. Adjust it so the noise channel is set to the left speaker and other notes to the right by checking boxes. You’ ll see a hex value “78” was generated for the config we chose (the effect slot now reads 878). Now play your song while wearing headphones. The hi-hats should come through one ear in your headphones and the rest on the other side. Now add a 8FF a few lines lower. Play the song. You’ ll hear the stereo resets. If you click the 8FF the UI shows that it means all channels on all stereo. This is

just one example! The entire list of effects for hUGEDriver emulator are here: <https://nickfa.ro/images/HUGETTrackerManual.pdf> on page 21. # Effects Terminology If you ask for help on the Discord, those using hUGETTracker and other trackers might refer to an effect as Cxy or 8xx: that means you put the right letter and then two hex values in the effect slot to activate the effect. The xx values means the two digits are evaluated together as one parameter. The xy means each individual digit represents a different parameter. Each effect has its own parameters. We will get into that now. # Effects That Impact Sound of Individual Channels ## **Set Volume (Cxy)** Use this to make a note quieter or louder. **Duty channel:** The easiest way to use this is to keep x at 0 and change y, which will play the note's default envelope at the volume level of y. See the sample uge at the end to hear the fading effect of reducing C values. **Wave channel:** The wave channel only has 4 volume levels and no envelope, so only these values listed below will have an impact. The values in between will just cause the same change as the value closest to it. **C00** – wave channel is silent **C03** – wave channel plays at 25% it's full volume **C07** – wave channel plays at 75% it's full volume **C0F** – wave channel plays at full volume. On the wave channel you can put a Cxy without triggering the note (meaning it can be on a line by itself without a note). Changing wave channel volume with Cxy can cause a clicking or popping sound on actual hardware. This can be minimized by using multiple instruments with the same wave and different volume levels. When possible this is a better solution and leaves you free to apply other effects to the wave channel independent of the volume. **Noise channel:** Cxy doesn't really work on the noise channel as you might expect. You will lose your envelope if you use it. It's better to use multiple instruments with the same envelope and different starting volumes. ## Arpeggio (0xy) The best example of an arpeggio is when you get a fire flower or mushroom in Super Mario Bros. It's a super fast progression of notes that happens in the space of a single tick. It is very common on Atari and Commodore 64 songs and is used a lot by some NES and Gameboy composers to allow you to "fake" three-note chords. Many will refer to them as "arps" for short. The parameters are how many notes up from the starting note it should play. x comes first followed by y. The y value can be higher and go back down. You can put multiple 0xy in successive ticks to make more changes in the notes. You can use them on duty and wave channels. You can also put them on ticks with no notes being triggered and they'll be applied to the existing envelope. They don't really work on the noise channel. Here are some common useful values: * **07C** – jumps up to a major harmonic note (7) and then the original note but an octave higher. This is a good starting out arpeggio to try. * **037** – minor key chord. Sounds scary or moody. * **047** – major key chord. Sounds happy. * **057** – Neutral chord. Sounds more expansive and unresolved. ## Note Cut (Exx) A very simple one. If you want to have a quick, short note this allows you to cut the note off (mute it) very quickly. Start with E02 and experiment with the length to get the sound you want. It usually sounds clearer to use an instrument with a short length instead of using the note cut effect. ## Portamento Up (1xx) Some physical instruments have the ability to have a sound bend smoothly from one note to

another. This is string bend on guitar, or slide on a violin. Synthesizers often have a wheel on the side to do this too. 1xx can be used when a note starts or at any point when it's playing. The higher the value, the faster the note goes up. You can use many 1xx in a row to make an ascending tone like a car engine charging up. This is another one that doesn't really work as expected in the noise channel. ## Portamento Down (2xx) Similar to 1xx, 2xx makes the note bend downward, like that awesome moment before the Mega Man 2 title music transitions from the slower intro music. 2xx is also our secret weapon for making bass kick drums. # Effects That Impact the Entire Song **Set Panning (8xx)** As we saw in our first example how panning lets you set the stereo for all channels. It's important to note you can put an 8xx in any channel and impact other channels. For example if you have lots of effects in duty 2 but also want to change its stereo, the 8xx commands can go in any other channel and still change duty 2. This is kind of a weird thing since most effects impact the channel they're on, but it lets you combine stereo changes with other effects on the same channel. The Gameboy sound chip only lets you play both channels or a hard left or right pan, meaning nothing panned left can be heard out of the right speaker or vice versa. You can also use 8xx to cut a channel without stopping its note from playing. **Set Tempo (Fxx)** You can speed up or slow down a song using Fxx. In most songs you probably just want a Fxx at the very first tick of the first pattern in order to set the tempo for the entire song. You can also use it to make a half speed part of a song (slow part) or speeding up sound (like Sonic's running out of air jingle). You can also use many of them together to create swing grooves. Which gives songs a jaunty, loose rhythm feel instead of a rigid techno-like beat. **Jump to Pattern (Bxx)** The Bxx lets you jump back to a previous part in your song. This is useful for making loops. You can make a song intro in pattern 1 and then have 4 patterns that make up the song. At the end of the 5th, a B02 in the final tick will send the song back to pattern 2 without playing the intro again. Note that the tick with the Bxx plays before the jump happens. **Pattern Break (Dxx)** Pattern break tells the system to stop the pattern it's on and go to the next one starting on tick xx. You can use this to make non 4/4 tempos, for example a waltz will jump forward to the next pattern after the 17th tick. There are other effects available but we are going to stick to these for now. In the next lesson we will add effects to our tutorial demo song. You can download this uge to hear all the examples shown in this article. In the next article we will start applying these practically in our demo song! Download Effects UGE file

This article is part of a series: GB Studio Music Beatscribe

Beatscribe

Beatscribe has composed and produced soundtracks for games on Nintendo DS, Nintendo Switch, PS4, Xbox1, Sega Genesis and numerous iOS and Android games for over a decade, but his true passion is creating epic moods on ancient hardware like the Game Boy and NES. ### Share this:

- [Click to share on Bluesky \(Opens in new window\)](#) Bluesky

- Click to share on X (Opens in new window) X
 - Click to share on Facebook (Opens in new window) Facebook
 - Click to share on Mastodon (Opens in new window) Mastodon
 - Click to share on LinkedIn (Opens in new window) LinkedIn
 - Click to share on Reddit (Opens in new window) Reddit ### *Related*
-

GB Studio Music Lesson 7: Special Tricks Part 1

This article will cover some cool tricks you can do with GB Studio tracker and hUGEdriver in order to get the most out of your songs. Make sure you’ve reviewed the effects commands in the previous articles before attempting these tricks.

To hear some of these tricks in action, we’ve provided example tracks that you can download below. Download [Example Tracks ## One-Time Intro](#) You know in Super Mario, Sonic or Megaman X games how there’s a little intro to the song that never plays again after the song gets going? You can do this with GB Studio UGE songs as well.

First make your song as you normally would but leave the first pattern blank. Once you’re done, go back and make an intro. In the download, **demo-intro.uge** has a drum intro and roll into the sample song from previous lessons. The music file “demo intro” displayed in GB Studio’s music tracker. If you play it like this the intro will play again when it reaches the end of pattern 2. But if we had a B02 at the end of pattern 2, this tells it to jump to pattern 2, effectively bypassing the intro, which now will only play once. You can make the intro as long as you want. For example, if the intro is 4 patterns long, just put B05 at the end of the song.

If your intro is short, perhaps just a few notes of a drum roll, you can use a D01 in the last row you use to skip ahead to the main song too. **## One Time Play Jingle** Sometimes we want a jingle to play once and then for the sound to sit quietly until the user does something. A great example is a game over jingle, like **sportsgameover.uge**, from my free jingle pack. It’s been included in this article’s download too.

Basically the same thing is happening as the One time intro but pattern 2 is silence. An image of GB Studio’s music tracker, showing the silence pattern from the “Sports Game Over” example song. You can even combine a jingle with a loop to do an RPG style post-battle victory plus a little loop like in **free-victory-loop.uge**. **## Swing Beats** For jazz, swing, funk, and hip hop, a swing groove is often used. A Swing Groove means some notes are a tiny bit longer than other notes. It can be a subtly funky swagger or a jagged choppy rhythm that somehow stays together. You can do these tricks but temporarily changing the tempo with a Fxx command and then putting back on the next beat. This example showed a common swing beat. By putting the two F values

further apart you can make a more subtle or more powerful swing groove. An example of a swing groove written in GB Studio's music tracker. You can hear this in **swing-demo.uge**.

If you haven't already, be sure to download the examples below to hear some of these great tricks for yourself! There are lots more tricks we will cover in part 2 of this article. Download Example Tracks

This article is part of a series: GB Studio Music Beatscribe

Beatscribe

Beatscribe has composed and produced soundtracks for games on Nintendo DS, Nintendo Switch, PS4, Xbox1, Sega Genesis and numerous iOS and Android games for over a decade, but his true passion is creating epic moods on ancient hardware like the Game Boy and NES. ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### *Related*

GB Studio Music Lesson 8: Common Mistakes

We all make mistakes! When composing in a tracker, small mistakes can compound quickly into a huge mess, or at the very least a weak sounding song. Here are some common pitfalls to avoid. ## Imagining Your Own Tempo Remember when I said to do drums first? I was saving you from this: the most common issue new composers make in GB Studio tracker. Let's assume we are brand new and don't know about Fxx yet. I want a slow song, so I lay out these drums: An image of a song's notes displayed in the GB Studio tracker. Looks fine and sounds fine. But, when I reach pattern 2, notice the kick drum does not fall on the 1st row! An image of the GB Studio tracker, showing an instrument not appearing in the first row of a pattern. Sure I can continue down this path like this; my song will sound fine, but later if I want to reuse a pattern, I will have a giant mess. I also will have to use some special commands to make it loop smoothly. What has happened is when I hit the play button, I imagined a tempo that is different than the actual one.

Let's fix it up: An image of the GB Studio tracker. The 8th and 16th rows are highlighted. Now that I have the F command slowing the song down instead, I have the same thing but each pattern ends on the 64th note and starts over with a kick on the 1st down beat like it should. This is why you always start with drums. Even if you just throw down some simple 8th notes (1 and 2 and 3

and 4 and ...) it' ll keep you from imagining the tempo being something other than it is.

Notice the green circles: you can be sure you are doing things right if your "down beats" (the 1, 2, 3, 4) of your song land on the light gray rows in GB Studio tracker and not in between them. ### Putting Leads on Duty Channel 2 Most likely the majority of your songs will play underneath sound effects at some point. The DMG sound chip doesn' t have extra channels for FX so they have to butt in on your songs for a few milliseconds to play. The following video shows how disruptive the FX are when the lead is on channel 2. It gets destroyed by even the tiniest sound effect. Later in the video, notice how we put the fast repeating rhythm part on Duty CH2. Now you barely miss the tiny note that plays for a quick moment. **Video with subtitles: Video with no subtitles:** Use Duty Channel 2 for the following: * Echoes – you won' t miss them as much as the main melody. * Fast notes – they' ll come back on quickly. * Drum augments – In some cases losing the tom or kick drum is better than losing the bassline or melody. * Harmonies – again, a harmony dropping out still leaves you a primary melody to remember. ## Unexpected Looping Results All songs will loop. You need to check what happens when the final notes play and the song starts over. In the case shown here, the wrong bass note will play for a few moments when the song begins again. It' ll sound weird. Use an E00 to cut channels at the very end of the final pattern or at the start of the first pattern if a certain channel doesn' t play immediately. This also works for stereo. If you were bouncing the stereo a lot, put a 8FF or whatever is appropriate at the start of the very first pattern. ## Wasting Channels Here is a simple beat and reggae rhythm with bassline: An image of the GB Studio tracker, showing the notes for a simple beat and reggae rhythm with bassline. It sounds nice but the kick drum has taken up Duty1, the back beat "ska" guitar has taken up Duty2, the wave is doing the bass line, and the high end drums are coming from noise. There' s no room for a melody.

Now let' s look at the song sideways: An image of the GB Studio tracker, flipping the notes on their side for a visual example. Imagine it like a Connect 4 game. Can anything fall into an empty gap? If you said Duty2 then you' re right. In ska and reggae the guitar plays in between the down beats and therefore, there is no reason it can' t share Duty1 with the kick drum. Literally just copy and paste those notes over. An image of the GB Studio tracker, with arrows pointing to the 12th, 14th, and 20th rows. Now you have a full channel free for melodies.

If you' ve made these mistakes, don' t feel bad. This article comes from personal experience. Trackers are hard. Just don' t give up and keep learning!

This article is part of a series: GB Studio Music Beatscribe

Beatscribe

Beatscribe has composed and produced soundtracks for games on Nintendo DS, Nintendo Switch, PS4, Xbox1, Sega Genesis and numerous iOS and Android

games for over a decade, but his true passion is creating epic moods on ancient hardware like the Game Boy and NES. ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
 - Click to share on X (Opens in new window) X
 - Click to share on Facebook (Opens in new window) Facebook
 - Click to share on Mastodon (Opens in new window) Mastodon
 - Click to share on LinkedIn (Opens in new window) LinkedIn
 - Click to share on Reddit (Opens in new window) Reddit ### *Related*
-

Getting Started in hUGETracker

This series will cover the use of the hUGETracker software. hUGETracker is a standalone app native to the hUGEDriver which allows you to create music that is playable on the Gameboy. hUGEDriver is a powerful driver allowing you to compose dynamic Gameboy songs without eating up so many resources that it can't play music during a busy Gameboy game.

This article will briefly cover the installation and UI as well as taking a quick look at a sample song which explores what the Gameboy can do. This article is centered on using hUGETracker for GB Studio, but you can use it to create songs for GBDK, RGBDS and any other platform that can use .wav files. ## **Installation** hUGETracker is a cross-platform stand-alone tracker. This article will cover just the basic installation. If you are using a version of GB Studio that comes before version 3.2, you'll want to download hUGETracker version 1.0 beta 9.

This link will take you to the latest version to install for GB Studio version 3.2 or higher. Just unzip all the files into a directory of your liking, then run hUGETracker.exe. You can make a pinned or desktop shortcut if you'd like as well. ## **User Interface** If you've used other trackers such as OpenMPT or Famitracker, the hUGETracker interface will seem very familiar. We will assume you've never touched a tracker before, and start from scratch in this series.

Let's start with the basics. Download the `htt-demo.uge` here (the sample UGE was made in 1b9 version but its forwards-compatible with any other hUGETracker versions, once you save it in a later version though, you cannot go back), drop it somewhere on your hard drive and then go to File > Open... in hUGETracker and locate it. You should see something like the picture below. Don't be intimidated, we will soon learn what everything does and how it works. Before we go any further, let's learn what all the components are called. A screenshot of the hUGETracker interface, showing a song loaded into the tool. The main part of the screen (gray section) is the Tracker Grid. This is where you compose your song. Don't worry about what all the numbers and letters in the grid mean right now. Let's start by looking at the overall layout and how it works.

At the top you'll notice 4 columns labeled Duty 1, Duty 2, Wave, and Noise. These represent the 4 channels the DMG sound chip is capable of playing. We will look at them briefly here, but dive into more details later on.

Duty 1 / Duty 2 – These are simple Pulse channels. They make up the sounds you hear mostly as melodies and leads in Gameboy music. They can make a thin or fat simple 8-bit tone and have a few other features we will look at later on.

Wave – The Wave channel is what gives the Gameboy its distinct crunchy sound. More often than not the Wave channel provides the bassline or bass kick drum sound of a song. The wave channel is highly configurable and can make beefy noisy blasts or delicate smooth tones.

Noise – The Noise channel basically molds atonal static into sounds of drums, buzzing machines, explosions, bells, ocean waves sounds, and wind. You almost always use it for drums or special effects.

The lines below each channel represent a **pattern**. In hUGETracker, a pattern is a list of commands for a single channel. The four patterns will play simultaneously at the same time through the soundchip. We will get into the insides of a pattern in a moment.

Each pattern is made up of 64 rows. A **row** is a single moment in time in the song. It might start a note, stop a note, or make some modification to the song's properties. Each row's events across the four channels are executed simultaneously. The speed at which they occur is controlled by the tempo.

To the left of the Tracker Grid is the **Order Editor**. This is a high level view of your song. The highlighted gray background row is the row you're currently viewing in the Tracker Grid. If you click on another row, the Tracker Grid will display the patterns in that row. In the example on the screenshot, we are viewing the first order of the song, which contains patterns 0 1 2 and 3. The Order Editor's cells can be edited, you can change the number of a pattern to assign a different pattern to a certain channel for an order of the song. We will get more into this later.

Note: Although the UI sometimes refers to rows in the Order Editor as a "row," they are really called Orders. An Order is a set of 4 patterns that are played together.

Now let's dive into the internal components of a pattern. A screenshot from the hUGETracker user interface. The Note Cell, Instrument Cell, and Effect Cell are highlighted. Each of the four channels' patterns contain the same properties. The **Note Cell** tells the tracker what note to fire off when this row is reached. It might be a tone or a drum noise depending on the channel.

Next is the **Instrument Cell**, this tells the tracker what instrument to play the Note Cell's note through. We will learn about instruments later, but imagine them as a bass guitar, a drum, a trumpet, etc. The number corresponds to the instruments you see on the far left of the screen per channel.

Finally we have the **Effect Cell**. Effects do things like make a note quieter, change their stereo location, stop a note, or bend a note up or down. We will get way more into this later.

So to read things, let's practice, look at Row 20. When the tracker is playing a song and it reaches Row 20, it says "Ok, Duty 1, you play a F5 note out of instrument 1. Duty 2, you're gonna play a G#5 out of instrument 2. Wave channel, I need you to fire off a A#4 with instrument 4's wave. Noise channel, give me a D8 out of instrument 2." This all happens at once.

What does that sound like? Let's learn how to play a song now and examine its pieces. **## Playing A Song** At the top of the UI, you will see buttons that say Start, Cursor, and Stop. These control the playback of your song. Most composers will play chunks of the song, stop it, and make adjustments many many times (at least I do).

Pressing Start will start the song at the first pattern in the first row. Basically it plays the song from the beginning.

Pressing Cursor will start the song playing from whichever row is highlighted in the Tracker Grid. This may result in a song playing at a different tempo or not having all the notes playing until it has played for a few rows. It is handy when working on a long song and playing back just one section to review it.

Pressing stop will kill all sounds and put the cursor back where it was prior to pressing play or cursor.

Ok, now let's hear some sounds. Press play and listen to some funky goodness.

As you can see, the highlighted line in the Tracker Grid shows you what's playing and what you're hearing as a result. As the song progresses notice the Order Editor also updates the Tracker Grid to show whichever order is currently playing. When it reaches the end it'll start over. **## Muting/Solo'ing Channels** Now let's learn to isolate a channel. Isolating channels is useful when you're not sure what's causing an issue with your sound, or if you'd like to learn how another musician has compiled his channels to work together (it's not always obvious). In music production speech, muting a channel means it won't play along with the other parts of the song. Solo'ing a channel means the other channels will be silenced so you can only hear one.

Click the little sound speaker icon on all the channels except Wave. Your UI should now look like this: A screenshot of the hUGETracker interface, showing what happens when you isolate a channel. The red cross-out icon means the channel will not play when you press Play or Cursor. The data is still there, this is just for you to hear as you playback. Press Play now. You should hear the bassline with no other instruments.

Play around and solo each channel to understand how the song is composed. Hopefully you come to this conclusion:

Duty 1 – This plays the main lead. It has a kind of thick sound and is fairly loud. At the very end the pattern it plays some 8-bit tom tom drum notes in a little roll-styled bit.

Duty 2 – Duty 2 has a thinner sound than Duty 1 and is playing a harmonic version of the same lead. After a few orders, the harmony is dropped in favor of a quieter backing pad note which plays in a sustained way behind the lead of Duty 1.

Wave – The Wave is just playing the funky bassline. It is crunchy and curiously plays pattern 2 across all the orders.

Noise – The Noise channel uses a couple different instruments to produce a disco-styled beat. The beat pattern 3 also plays across all four orders. ##
When to use hUGETracker or GB Studio Tracker There are benefits to using GB Studio Tracker or hUGETracker depending on your needs. For short simple songs, and especially when you’ re just starting out, GB Studio tracker is a lot less confusing to use. In GB Studio Tracker the concept of a pattern is a bit different than in hUGETracker but we won’ t go into it here.

hUGETracker is better for songs that will be long and for which you desire to reuse some patterns along with other unique patterns within an order. It also has more export features you need if you aren’ t using GB Studio (export to .gbs, .wav, .c, .asm).

Outside of exporting to other formats, the honest truth is there is nothing you can do in hUGETracker that you cannot do in GB Studio Tracker. You just have a little less granular control over patterns and orders in GB Studio Tracker, but you should use whichever you prefer. ## **Next Steps** So, although we’ ve yet to put down a single note of our own design, we should understand fairly well how all the individual pieces come together. The next lesson will get into making your own song and first instruments as well as using the noise channel to make some simple drums.

This article is part of a series: hUGETracker Beatscribe

Beatscribe

Beatscribe has composed and produced soundtracks for games on Nintendo DS, Nintendo Switch, PS4, Xbox1, Sega Genesis and numerous iOs and Android games for over a decade, but his true passion is creating epic moods on ancient hardware like the Game Boy and NES. ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### *Related*

VGM Sound Effects Part 3

If you’ ve followed these tutorials up until now, you’ ve created an awesome “boing!” sound in Furnace Tracker. Now it’ s time to move it into GB Studio! It’ s not as straightforward as you might imagine. And we will take the time to demonstrate why. If you’ ve mastered using Furnace, you’ re already through the hardest part. ## Export to VGM Save your .fur file somewhere that you can retrieve it later. Once you export to .vgm, you cannot reload it into Furnace Tracker (it only opens .fur files, not .vgm). Many developers like to make a /src file in their GB Studio folder to keep files like .fur in. They aren’ t part of the build, but they are part of the project’ s DNA you might say.

In furnace go to File > Export VGM Make sure you uncheck Loop and are using Format version 1.72. Then click “Click to Export.” Save the file wherever you want but keep in mind it’ s not in its final format. You probably don’ t want to put it into /sounds/ in your project just yet! ## Removing Channel Kill Codes With VgmGBSTudionator If you were to drop the .vgm we just made into /sounds and use it in your GB Studio project, it would work, however it would kill the duty and noise channels everytime it plays, resulting in quite a jarring feeling. Our sound only uses duty2 and therefore should only need to interrupt duty 2 (which most Gameboy composers expect and make sure duty2 doesn’ t contain the songs main melody or bass).

To fix this, we have to remove some hex values from the .vgm file. This is a tedious process and if you have 20 sounds, you need to do it perfectly 20 times. Fortunately, PotatoTeto has made VgmGbStudionator – a handy tool which you should have already installed in Lesson 1.

Copy your .vgm to the clipboard and open VgmGbStudionator wherever you installed it. Inside you’ ll find an input folder, past the .vgm inside the input folder. It works in batches, so if you have other .vgm throw them in there too. Now in the main VgmGbStudionator folder, double-click the convert.bat to run it. Press any key. Now go to the /output sub folder ## Importing into GB Studio Copy the .vgm file(s) from the /output folder. Now go to your GB Studio project. Newer versions will already have a /assets/sounds/ sub folder. If you don’ t see one, don’ t worry, just make a sounds sub folder. Paste the .vgms inside the sounds subfolder.

Now open your project. You can trigger a sound using a Play Sound Effect command In this example I put one under an Attach Script to Button event so the “boing” plays when you press B. That’ s it! Now you can use .vgm sounds in your game! Remember if you didn’ t make the template right or made too long of a sound, it’ ll play too fast or play infinitely. ## Manually Removing Hex Kill Codes If you couldn’ t use VgmGbStudionator, you can manually remove the Hex Codes using a program like ImHex. Here’ s the hex chunk you need to remove. Just remove and save the file again as a .vgm (this is basically what VgmGbStudionator is doing). That’ s really all there is to it. If you can’ t get VgmGbStudionator going, I’ d almost recommend asking a

friend with a PC to do this for you as it's less hassle.

VGM Sound effects are very powerful and I'm really glad you can use them in GB Studio. Hopefully this series has helped you to get the basics down and you can start making some crazy sounds.

This article is part of a series: [VGM Sound Effects Beatscribe](#)

[Beatscribe](#)

Beatscribe has composed and produced soundtracks for games on Nintendo DS, Nintendo Switch, PS4, Xbox1, Sega Genesis and numerous iOS and Android games for over a decade, but his true passion is creating epic moods on ancient hardware like the Game Boy and NES. ### Share this:

- [Click to share on Bluesky \(Opens in new window\) Bluesky](#)
- [Click to share on X \(Opens in new window\) X](#)
- [Click to share on Facebook \(Opens in new window\) Facebook](#)
- [Click to share on Mastodon \(Opens in new window\) Mastodon](#)
- [Click to share on LinkedIn \(Opens in new window\) LinkedIn](#)
- [Click to share on Reddit \(Opens in new window\) Reddit ### Related](#)

VGM Sound Effects Part I

Making sound effects for the Gameboy is a complicated procedure. Since the Gameboy only has four channels, sound effects have to interrupt parts of the music. Being limited to using 1 to 3 channels of the sound, it can be very tricky to convey a certain action or sound successfully in such a limited medium.

This series of articles will cover how to create custom sound effects using the VGM format for GB Studio games. There are other ways to generate sound effects, but VGM has lots of benefits, including easy file management, extremely full-features effects, and even the ability to make longer than usual sounds. There's quite a lot to learn, but we will start from scratch so you can get up to speed. This first article will cover getting the right tools installed and how to start creating sound effects. The second will have an in-depth look into the effects and commands needed. Finally, we will look at how to ready your sounds for use in GB Studio. ## Get The Needed Tools There are two important programs you'll need to install in order to make your own VGM sound effects. There are various ways to create VGMs you can use with GB Studio. However, the path shown here is the easiest option if you want to avoid editing Hex code and dealing with surprises. You can do this project on either a Mac or Windows. If you have the option, however, Windows is preferred. ## Install Furnace Tracker If you are on Windows, download the latest version of Furnace Tracker and install it. If you are on a Mac, use Furnace v0.6.1 since the hex editing you'll need to do needs VGM exported from that version. We will dig into Furnace in more detail later. ## Install VgmGbStudionator By default,

the VGM that Furnace makes has some codes that kill all the channels at the start. This is bad for your game. It will mute all the music when a sound effect plays. PotatoTeto has made a nice little tool called VgmGbStudionator (the name was my suggestion, don't blame him!). This allows you to batch-strip all those kill codes you don't need. If you don't have a way to run it on a Mac, we'll cover how to manually remove the hex kill codes, but it's not fun.

If you can't get this running on a Mac or Linux box, you'll need a hex editor. I recommend ImHex. ## Setting Up A Furnace Template You will design your sounds in Furnace Tracker. This is a full-featured Tracker, so it can be overwhelming. For this reason, we will only teach you what you need to know for making sound effects.

Follow these steps to prepare a template. You'll only have to do it once. * Go to File > New... * Select Gameboy from the list (you can filter if needed) * Find the Speed tab in the upper right corner * Set the Base tempo to 640 * Go to Save.. and put the .fur file somewhere You'll want to start each new sound effect from this template. I usually open it and do a "Save As..." immediately, or else I end up overwriting it. ## Creating An Instrument When you build a sound effect, it is basically creating a very short song. If you're familiar with GB Studio tracker or hUGETracker you won't have any trouble getting used to Furnace Tracker. * Go to the Instrument Tab * Click + * Double click the instrument to open its window We will very quickly cover the settings of an instrument. ### Gameboy Tab **Volume** – The volume is the starting level of the sound.

Length – This is how long the sound takes to play. Imagine it as how fast you turn down the volume knob after hitting a note.

Sound Length – While Length is like a volume knob, Sound Length is like a Mute button. You can cut the sound off quicker than its natural fade if this suits you.

Direction – This basically says are you turning the volume knob up or down. It starts at the volume value and goes up or down at the speed of the Length. ###



Macros Tab

The macros tab has lots of things to further refine your sound. To edit one of the parameters click the icon next to them. You can then click the arrow button to make a sequence of parameters. In the picture above you can see a series of notes which change pitch up and down. You can click the ... to control how fast it cycles through the parameter changes. You can draw them with the mouse on the yellow graphic or write in the numbers below.

The Arrow Button **Arpeggios** – The classic up and down jagged pitch changing you hear in classic chiptunes. In the context of sound effects, it's great for making motor sounds or power up sounds.

Duty/Noise – This is one of the most important settings. Often you only need 1 slot here, meaning it will only change once. The settings work like this: ###
Duty channel **0 – 12.5% duty** – a thin sounding tone

1 – 25% duty – a middle ground tone with a hint of a second octave

2 – 50% duty – a softer, thick, round sounding tone

3 – 75% duty – a middle ground tone with a hint of a second octave

Waveform – This is used to change wave bank forms. However, it doesn't work in GB Studio, so just ignore this.

Panning – You can use panning in GB Studio but I don't recommend it because it can throw off the songs stereo. You can either choose left, right or both.

Pitch – Pitch is another very useful one, this is for portamento style pitch bends up or down, instead of jagged arpeggios. ## Making Our First Sound Let's use what we learned to make a classic cartoon "Boing!" sound. Your goal is to make a short length instrument that starts loud and gets quiet quickly. It should be 50% duty so it feels rubbery and thick and has a bending pitch for the "boing" sound.

Set up the "Gameboy" tab as shown: Set up the "Macro" tab as shown: Also

click the ... and set the step length (this slows down how fast it goes through the notes). ### Laying Down Notes Now that your instrument is created, you need to learn how to lay down the notes. This is done in the main tracker area: Press Space Bar to switch between read and write mode. The blue bar means you are in read mode (this is used when you don't want to accidentally lay down a note). The red bar means you're in editing mode. Press buttons on the keyboard to make notes. When in read mode you can test what the notes sound like before writing them. Press X to put down a D-3 note. You can change the octave at the top center area if you need.

Now press Enter to hear the sound play. You can download the .fur file here which has what we built in this article. [being.furDownload There!](#) You've got your first sound effect made. There's still a lot more to learn. We will get to that in the next two articles.

This article is part of a series: [VGM Sound Effects Beatscribe](#)

[Beatscribe](#)

Beatscribe has composed and produced soundtracks for games on Nintendo DS, Nintendo Switch, PS4, Xbox1, Sega Genesis and numerous iOS and Android games for over a decade, but his true passion is creating epic moods on ancient hardware like the Game Boy and NES. ### Share this:

- [Click to share on Bluesky \(Opens in new window\) Bluesky](#)
- [Click to share on X \(Opens in new window\) X](#)
- [Click to share on Facebook \(Opens in new window\) Facebook](#)
- [Click to share on Mastodon \(Opens in new window\) Mastodon](#)
- [Click to share on LinkedIn \(Opens in new window\) LinkedIn](#)
- [Click to share on Reddit \(Opens in new window\) Reddit](#) ### *Related*

hUGETracker Lesson 2 : Noise Channel

So, we got used to the hUGETracker interface last time. Now it's time to actually put down some notes and get more familiar with the tracker. The last article showed you a sample song. Now we're going to start making one from scratch.

Start a new song in hUGETracker. There are lots of ways to write a song, but I like to start with the drums. The reason for this is that it's very easy to get off-tempo in a tracker, meaning you're imagining a certain tempo when, in fact, it's not aligned to the grid. Drums are also an easy place to start as they're mostly atonal; you don't need to worry about such concerns as keys and chord progressions. ## Important Terminology Here are some important terms you'll need to understand as we delve into these lessons:

.UGE

A .uge file is a song basically; it's a special file that GB Studio Tracker, the

hUGEDriver, and other music trackers like hUGETracker can read and modify.

Tick

A tiny portion of time that passes as a song plays.

Tracker

A tracker is a special editor for making, modifying and playing back music. Like Microsoft Word can open a .docx file, trackers like hUGETracker and GB Studio Tracker can open .uge files.

Tempo

Ever heard a song that starts with someone yelling “1, 2, 3, 4” ? This is the invisible clock for the song. Everything is playing at the speed of the 1, 2, 3, 4, on the numbers or on divided increments of it.

Channel

One of the four Game Boy sound types that the sound chip can generate as explained previously.

Pattern

A set of 64 notes (numbered 0 to 63) that the four channels will play together. Think of it like the holes an old player piano reads.

Instrument

A collection of settings for a certain channel type which produce a sound. Settings are things like the envelope, the volume level, wave form, etc.

Envelope

Imagine that each time a note is played, an invisible hand turns the volume knob up or down on that instrument, this is the envelope.

Attack

The start of the envelope or how fast the invisible hand turns the imaginary knob up to full.

Decay

The end of the envelope or how fast and when the invisible hand turns the imaginary knob to 0.

Effect

Special commands that change a note in some way, they change its stereo panning, bend it up or down, or change its volume. Some of them impact the entire song, like ending a pattern early or jumping back to a previous pattern. ## Making Your First Instrument Making an instrument in hUGETracker is fairly simple. A chiptune instrument is much like a real one. It’s a configuration that generates a certain character of sound. Unlike a real instrument, a hUGETracker instrument has just a couple of settings.

The drums – as we learned in the last article – come mostly from the noise channel. Instruments are divided into three groups on the left hand side. Duty, Wave, and Noise – 15 instruments for each of the 3 channel types. By default,

there are some configured as duty and wave instruments. But we are going to make our first noise instrument from scratch. Expand the noise instrument section, and double click on number 1. An expanded view of the hUGETracker user interface. ### **Instrument Settings** These have to do with how the instrument is identified and sounds.

NOTE: The other channels have slightly different controls. We will cover the unique parts when we reach them in the tutorial. Most of the settings here apply to all instrument channel types.

Channel and Instrument

These are basically navigational shortcuts you can use to change what instrument you're editing or what channel type. Any changes are automatically saved when you go to the next one.

Name

This is to help you find the instrument when composing. Meaningful names go a long way in helping you stay organized.

Length

Designates how long the note will play, also known as decay. It may seem backwards but it would seem to imply how much of the original envelope gets trimmed off (so higher value means less of the original note plays).

Test Octave

Lets you preview how the instrument will sound at different octaves.

Enabled Subpattern

Click this if you want to use the subpattern (will get into what this is later).

Import and Export

You can save an instrument as a stand-alone .ugi file. You can generate these with export and bring them in with import. If you're using beta9 for older versions of GB Studio there is a bug where the instrument settings won't update when you import. You'll have to go to another instrument and come back to see the settings (but it does work).

Envelope Settings

As you adjust the envelope settings and (and also length) it will change the shape of the visualization of the instruments envelope.

Start Volume

This is the level the sound is playing at when it is first triggered.

Direction

Once the note is played you can "turn the volume" up or down (or leave it alone). This designates if the sound is going to move up or down in volume as it plays.

Change

This is the degree of change over time of the envelope, or how fast the "volume knob" is turned up or down."

Subpattern

Allows you to configure a string of modifications to the instrument that fire off each time it plays. ### Noise Specific Settings **7-bit Counter**

Selecting this changes the noise from more atonal white noise to a tone like sound. You check this for buzzy and distorted sounding instruments.

Let's make our first instrument. Name it closed hat and configure it according to the image shown here: An image of an instrument configured in hUGETracker. If you test octave 8, you should hear something resembling a closed hi-hat.

Playing and Placing Notes

Now that we've made an instrument, the time has come to finally actually lay down a note. Go to the patterns tab, and pull down the instrument drop down, and select Noise 1: closed hat. Now position the cursor in the first row on the noise channel.

When we want to put down notes we use the keyboard. Once you're in a cell in the tracker, the keyboard becomes your piano keys. Here's how it's laid out: An image of a music tracker keyboard. If you press "Q", you'll probably hear a barely audible blip of rumbling buzz or low static. Doesn't sound great. That's ok. This is because the noise instrument has the most shape and sound at its higher level octaves. To change the octaves of your keyboard, pull down the octave box and select 3. Now try pressing Q, A, and then Z. You'll hear each one is an octave higher starting at octave 3. When you pressed Z, it probably sounded closest to a closed hi-hat.

Try pressing C or V. You'll notice these sounds are closest to the hi-hat sound we want. Try going all the way up the bottom level of the keyboard, you'll notice after pressing V, the higher notes basically just end up being low static again. This just means you passed the range of the noise channel so it went back to the bottom. If you look at the sample song, you'll see the first two notes are a D8 (press C) and a D#8 (V) with a blank tick in-between. But before you can go further, you're going to need the open hi-hat and the snare instrument made. Use what you learned to build them yourself based on these pics: **Open Hi-Hat** An Open Hi-Hat configured in hUGETracker. **Snare** A Snare configured in hUGETracker. Now that we have all three, lay out the pattern as shown here, changing instruments when needed. A music pattern configured in hUGETracker. Notice that if you're doing things right, most of the notes will be on the highlighted lines or directly between them. You might notice your song sounds slower than the demo. Don't worry about that, we will come back to that later on. You should have what's contained in this file at this point. Demo Lesson 2.ugeDownload For now our song only consists of one order with one pattern made for the noise channel. Don't worry about that either. We will slowly build on what we know to create a full song as the series continues.

This article is part of a series: hUGETracker Beatscribe

Beatscribe

Beatscribe has composed and produced soundtracks for games on Nintendo DS, Nintendo Switch, PS4, Xbox1, Sega Genesis and numerous iOS and Android games for over a decade, but his true passion is creating epic moods on ancient hardware like the Game Boy and NES. ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### Related

|| Effect Value | C || Parameters | X = Set attack or decay (see UI) or leave 0 to not touch the envelope. Y = Set starting volume level. || On Trigger | ☒ || On Playing Note | ☒ || Works on Duty | ☒ || Works on Wave | ☒ || Works on Noise | ☐ ||

A note which echoes away quieter and quieter. On the duty channel the volume effect only works when triggering a note. That means the effect can't be on a line without an accompanying note (it'll just be ignored). Use the UI to understand the parameters easily. Usually on the duty channel volume control is used to make an echo effect without creating a second instrument. Or morphing the envelope setting of a series of notes for the same instrument. #### Wave channel || || ***

hUGETracker Lesson 5: Effects - Portamentos, Volume Control and Note Cuts

Effects Part 1: Portamento, Note Cuts and Volume Control.

This is the first in a three-part series on effects. Effects are a huge area of composing in hUGETracker. While most effects only change minor things, using them in conjunction takes a song from feeling flat & dull to bursting with character. I recommend focusing on learning one effect at a time. Each works slightly differently. ### Using effects A screenshot showing how effects are placed in Hugetracker. Effects are placed with the keyboard just like notes. However they go in the 3rd column of each channel. Each effect has a hex value that identifies the effect type, and then two hex values that are the parameters of the effect. Sometimes both values are used together, and sometimes each one controls something different. Some effects work differently on noise or wave channels too. Some only impact a note that is triggered (started) on the same row, some modify already playing notes on the fly. This will also be covered. An effect used On Trigger. On Trigger – An effect played on a triggered note. It starts working as soon as the note's row is played. An effect used On Playing. On Playing – Multiple effect calls on an already playing note. They will be applied to the already in progress sound as the row they are on is reached.

So, what are the parameters and are they hard to use? Some of them are, they require hex math and bit masking, but the good news is hUGetracker gives you a wonderful UI interface for each effect! After placing an effect value, double click it to bring up its special UI. Screenshot of hugetracker' s effects editor UI. ### How we talk about effects New folks are often confused in #audio-help by the way we talk about effects. If someone tells you: “use 8xx on the noise channel,” it means use the effect with values that suit you on that channel. This is also how they’ re explained in the hUGETracker manual.

If someone says “Cxy” that means the two hex values are separated, if they say “Fxx” it means the two values work together.

Each effect type will be explained along with a screenshot of an example usage you can try for yourself. We will dive right in with one of the easiest ones. ##### Note cut ||| *** ## IV. Art & Visuals - Animating Background Tiles (3.0.3) - Animating Background Tiles (3.1) - Basics: The Sprite Editor - Color Reproduction on GBC Compatible Devices and Emulators - Converting Images for GB Studio - Creating “Name Labels” for Dialogue Boxes - Creating Multiple Dialog Frames using UI Palette Tricks - Development Workflow – Chapter 9: Creating a Color Palette (Part 1) - Development Workflow – Chapter 9: Creating a Color Palette (Part 2) - Lights Out - DMG Custom Color Palettes Tutorial - Managing Sprite Tiles - Sprite Style Differences in Game Boy and Other Consoles - Understanding Fonts

Animating Background Tiles (3.0.3)

Note: *At the time of writing GBVM scripting is not fully documented and some of my statements regarding GBVM scripting might be incorrect or incomplete. This tutorial was developed using GB Studio V3.0.3 and is applicable to that specific version. If you are using GB Studio V3.1, please refer to this article instead.* This tutorial is based on NalaFala (Yousurname)’ s Background Tile Swapping Guide for GB Studio 3. Many thanks for providing the basis. To set up what we’ re going to try and achieve in this article, let’ s take a look at DuckTales 2’ s Niagara level. It features multiple large animated waterfalls that go far beyond the sprite limit. The Game Boy has some neat tricks, so how did they achieve this? Knowing that, how can we recreate it in GB Studio?



Ducktales on GB' s Animated Waterfall – too much animation for sprites! Let' s start by breaking down the effect: 1. There are multiple animated waterfalls concurrently on screen. 2. The player character renders in front of the waterfall without flickering. 3. Given it is not possible with sprites, then it must be using background tiles. 4. What GB Studio features can we use to animate background tiles? With this in mind I' ll jump right to it: The animation is created by swapping background tiles for another in memory, and I' ll explain how you can do this in GB Studio. ## They are 5 basic aspects of tile swapping I' ll be covering: 1. The function used to swap a tile. 2. How to create a simple animation. 3. The impact and limitations of replacing a tile. 4. How to replace a single tile. 5. Replacing multiple unique tiles. ## Additionally these GBVM Script features: * Accessing local and global variables. * Updating local and global variables. # GBVM Script Event While this might be considered a given to write code using GBVM add the **[Miscellaneous: GBVM Script]** event.



The event interface is basic (not the language), all you get is a text box where

you can write whatever GBVM commands you need. I would love to see future versions implement a simple code auto complete, otherwise known as intellisense in a traditional IDE. GB Studio Central has a great primer Understanding GBVM but it is worth reading the GBVM Command Reference as it provides documentation non present in the official GBVM source file. For this tutorial we will be using these GBVM commands * **VM_REPLACE_TILE_XY** * **VM_SET_CONST** * **VM_POP** I won't be explaining all of them in detail for this article, but if you have more questions about them it's best to asked them in the *#gbvm-help* Discord channel. The GB Studio Discord is full of helpful members, and it's a great resource and you'll see me there often asking question after question. ## Tile Swapping Functions There are two functions within GBVM that can be used to swap a tile for another in-memory: * **VM_REPLACE_TILE** * **VM_REPLACE_TILE_XY** Of the two, I prefer to use **VM_REPLACE_TILE_XY**, since it partially solves one of the challenges, locating and defining the target tile to swap. The syntax, or "way to write" the command is as follows, which each part explained. For this tutorial, I refer to the tile currently on screen as the "target" , and the tile which will be replacing it as the "source" .

```
VM_REPLACE_TILE_XY X, Y, TILEDATA_BANK, TILEDATA, START_IDX
```

```
||
```

Basics: The Sprite Editor

GB Studio 3.0 introduced many new features. One of the most noticeable features is the new Sprite Editor. Those who are familiar with previous versions know that users used to be relegated to following strict sprite sheet guidelines in order for the engine to determine if the sprite was stationary, able to point different directions, or move around. These limitations were good for beginners, as they were forced into a narrow workflow and having a limited scope helps ensure that first time devs would be more likely to successfully complete projects. It became quickly apparent that these limitations also limited what kind of sprites can be made, and by extension what games were possible to create. Some clever workarounds and tricks were figured out by the community to include things like large sprites or 3 frame walk cycles, but all that is a thing of the past with the **Sprite Editor**. Users are able to create complex animations, large sprites, and many different states that help define how a game looks and feels. Many devs have already dove in and figured out the nuances of the editor, but if you're just starting out, or maybe you wanted to learn more about it from the ground up, it's time we properly introduced you to the features of the sprite editor and how to use it. ## Before Starting It's important to note that while the Sprite Editor is a great tool and can really unlock some awesome game development tools, it's not required to make a game. If you are a fan of GB

Studio’s default sprite behavior, you can keep using it without any detriment to your development. If this is your first game, it’s probably best to stick to the default sprite settings too. For a refresher, read the official GB Studio Docs to learn about “Simple Sprites” and the four types you can create. You can also read on in the documentation for how the Sprite Editor handles more complex sprites, otherwise continue reading our tutorial here. We’ll be using the sample project, which is a great way to learn what’s possible. ## Sprite Sheets The first thing to understand is how the sprite assets are determined by GB Studio. Users who are familiar with previous versions of GB Studio or other development tools might assume that the best way to manage different actions like running, jumping, and attacks would be created in different sprite sheets to keep things organized. In GBS 3 however, this is handled by *animation states* (more on that later). Ideally all the different poses, actions and variations of a sprite should be included on one large sheet. Think of each sheet like the pieces of a doll or action figure that you will be putting together. You can really break things down into small sections to build within the editor, or you can create whole sprites and animation frames. GB Studio will try to optimize the tiles used by each sprite, so duplicate or empty tiles and frames shouldn’t take up more resources. Let’s start our understanding of the Sprite Editor by looking at the Sample Project. You can create one in GB Studio by going to the **File Menu** and Selecting **“New Project”** (this dashboard also shows up when you launch GB Studio), then make sure the “Sample Project” is selected in the Template section.

GBS 3’s template selection menu Now head to the View Menu at the top of the screen and select **“Sprites “**. On the top left side of the screen, you’ll see a whole bunch of sprites listed in the *Sprite Navigator*. Each one of these corresponds to a PNG file that is stored in the project’s `/assets/sprites` folder. If you want to create a new sprite, you will need to create a whole new sprite sheet PNG file or duplicate an existing one. You cannot simply add a reference to an existing sheet, which is contrary to how scenes and other assets work in the Game World view.

Every PNG file here corresponds to a sprite in the Sprite Editor. It’s important to still work within the parameters of how GB Studio handles sprites, so make sure any sprite sheets you create use the correct colors, are at least 16 x 16 pixels in size and that the sprite sheet’s height and width is divisible by a value of 8 to line up with the 8 x 8 tiles. Let’s take a look at the **player_platform** sprite in the sprite list on the left. Clicking on it will bring up the *Animation Navigator* on the bottom left, the Canvas, Tile Selector and Frame Timeline in the middle, and a context sensitive details window on the right known as the *Editor Sidebar*. The panes in this window can be stretched and moved around to best suit what mode you are working in.

The Sprite Editor Window ## Animation States You can start navigating around and see how things work, but I find learning the tools are best understood by building something new, so let’s try that now. We’re going to change the way animation states are handled, so we can build some new states from scratch.

Notice the *Animations Sidebar* on the left. It has only Animation states for the Right Direction. These specific states are the default for the Platformer game style, and we can see that is the *Animation Type* selected on the right sidebar. There's a checkbox under the Animation Type that says **"Flip 'Right' To Create 'Left' Facing Frames"**, uncheck that box and you will see the Animations Navigator suddenly includes three new states: **Idle Left**, **Moving Left** and **Jump Left**. The default behavior is to simply flip these frames in game to save resources, and that's perfectly fine for most games. There are some games, however, that require separate sprites for facing left and right, so by disabling that checkbox you can now create your own.

Samus' weapon is on her right arm, so her left and right sprites are different. Select the newly created **"Idle Left"** Animation State from the navigator on the left and notice that both the *Canvas* and *Frames Timeline* are empty.

The "Idle Left" animation state has no frame or canvas data. ## The Sprite Canvas It's time to populate those areas. The Canvas is a preview of what the player will see in game, while the *Tiles Selector* shows what tiles are available to place in that Canvas. You can manipulate the size of the canvas on the top right along with the collision box, but this will remain consistent across all animation states. Move your mouse around the *Tile Selector*, and you will notice that a shadow appears on it, always two tiles high. This is important to understand. While the Game Boy works with 8×8 tiles, sprites themselves can be 8×8 or 8×16. For both simplicity and to save on resources, GB Studio only allows 8×16 sprites, so anytime you select from the Tiles area, it will have to be a multiple of that. Let's do this now. Click and Drag around the first animation frame to highlight it in a red box. Make sure you start on the top left of the first sprite image and drag to down to the bottom right. Once that area is selected in the Tiles, move your mouse on top of the *Canvas* and you will notice the selection appears allowing you to position it. Click on the canvas to set it.

Applying your selected tiles to the canvas field. Notice how the *Canvas* displays the sprite as four separate boxes? This is actually what's happening. You've selected 4 8×16 sprite tiles from the *Tile Selector* and placed them adjacent to each other on the canvas. You've just set your first frame! Now we have one problem. The sprite is supposed to be facing left. You could open an image editor, flip the frame, and append the sprite sheet with it, but GB Studio already has this built into the *Sprite Editor*. If you clicked off the canvas (like I did above) you'll want to re-select all four sprites in the canvas first. Notice if you try to click and drag like you did in the *Tile Selector*, it won't work as expected, you will instead move one of the 8×16 tiles in the canvas. Instead, hold Shift on your keyboard and click on the four different sprites to select them all. After you click the first one, you'll notice the *Sidebar* to the right changes, and new buttons appear. One of the buttons looks like a paper airplane pointing up, that button will mirror the sprites on the Y axis. Click that to flip the sprite.

Flipping sprites across the horizontal axis ## Frames Timeline You should now know how to pull frames into the canvas and manipulate them. There are

many more buttons and features of the sidebar, but we’ ll have to save those for another time. For now, let’ s continue building this Idle Left animation using the *Frame Timeline* on the bottom. If we examine the Idle Right Animation State, we can see that the Idle Animation is actually 6 frames in length. If we number the first three poses in the “**player_platform**” sprite sheet from left to right, then the order of the frames in the timeline are 1, 1, 2, 2, 1, 3 – the player stays still for 2 frames, drops their head slightly for 2 frames, returns to the first pose for 1 frame, then blinks their eyes for 1 frame. It is possible to speed up and slow down animation within the GB Studio game world with an event, but there is a limit to how slow things can animate, and if you have uneven animation (like above where the pattern is not evenly spread) then duplicating frames is necessary. Rather than doing all the work of adding another frame and placing the same sprite sheet pose as before, you can simply copy and paste the first frame in the *Frame Timeline*. Click on the first frame to highlight it. Then either go to the **Edit** menu and select **Copy** then **Paste**, or use the keyboard shortcuts Ctrl-C, Ctrl-V (Command-C, Command-V on Mac) to duplicate the frame. You now have a perfect copy of that first frame. For the next frame, you’ ll want a blank canvas to start with, so press the “+” button in the *Frame Timeline*. Follow the steps as above, but this time select the second pose where the player’ s head is slightly lower. Then go ahead and copy and paste that frame too.

Adding (and flipping) additional frames to the Frames Timeline You can also copy and paste frames out of order. So select the first frame (which is needed for the fifth frame now), copy and paste it – it will automatically go to the last frame position, which is where we need it. If you needed a pasted frame somewhere else, you can simply drag and drop it in the timeline. You can also delete unwanted frames as well using the “**Delete**” key on your keyboard. The last frame we need is the blink, so go ahead and create a new empty frame with the “+” button again and add the appropriate pose to the canvas.

Adding the final “blinking” animation frame for the Idle Left animation You can now preview your work using the play button in the top left of the canvas viewer.

Previewing the Idle Left animation in its entirety (on loop) We’ ve only scratched the surface of what is possible with the Sprite Editor, but at this point you should have a basic understanding of what the workflow is like to create large and complex sprites in your game. 1. Create a Sprite Sheet and add it to the assets folder. 2. Go to the Sprite Editor and select the sprite. 3. Select the type of animation in the sidebar to set the default animation states. 4. Build your animations in the canvas using the tile selector. In the future, we’ ll go over some of the more advanced features like creating custom animation states and calling them in the game, as well as creating multi color sprites and layering sprites. For now, go to town and have fun thinking about how you can use this powerful tool! Emi Paternostro

Emi Paternostro

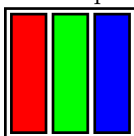
Audio Engineer, Mac Technologist and Video Game Developer. Managing Editor of GBStudio Central. (he/him)

proximitysound.com ### Share this:

- [Click to share on Bluesky](#) (Opens in new window) Bluesky
- [Click to share on X](#) (Opens in new window) X
- [Click to share on Facebook](#) (Opens in new window) Facebook
- [Click to share on Mastodon](#) (Opens in new window) Mastodon
- [Click to share on LinkedIn](#) (Opens in new window) LinkedIn
- [Click to share on Reddit](#) (Opens in new window) Reddit ### *Related*

Color Reproduction on GBC Compatible Devices and Emulators

Color reproduction on the Game Boy line of consoles is important for many developers, yet there is little discussion on this topic. I want to explain color differences between devices and emulators compatible with GBC software in this article, and why it's something that should matter. Let's begin with the basics: how do color screens display colors? If you look at your screen closely, you can see three colors inside a pixel, red, green and blue. These are known as

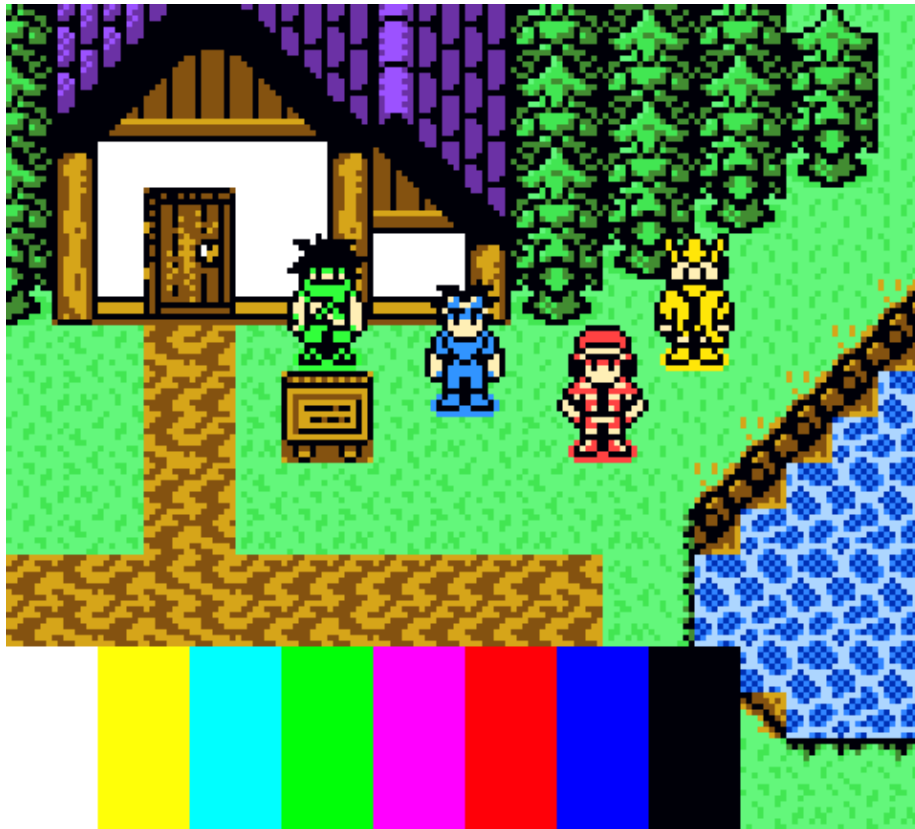


additive primary colors.

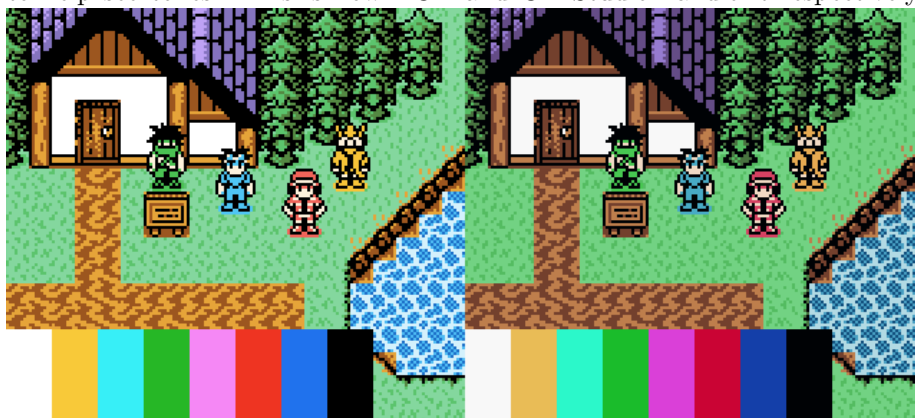
The three RGB sub-pixels It is possible they might be arranged a bit differently, but they're always the same colors. They can look rather simple, but they're the only colors necessary to create thousands of other colors in a display. Each sub-pixel can output a range of luminosity independent from each other, the result of which displays a specific color pixel (This luminosity range is usually 0-255 in most computer monitors, 0-31 on the GBC). All color displays use this method: your computer monitor, your phone, your TV, and of course, your Nintendo handheld.



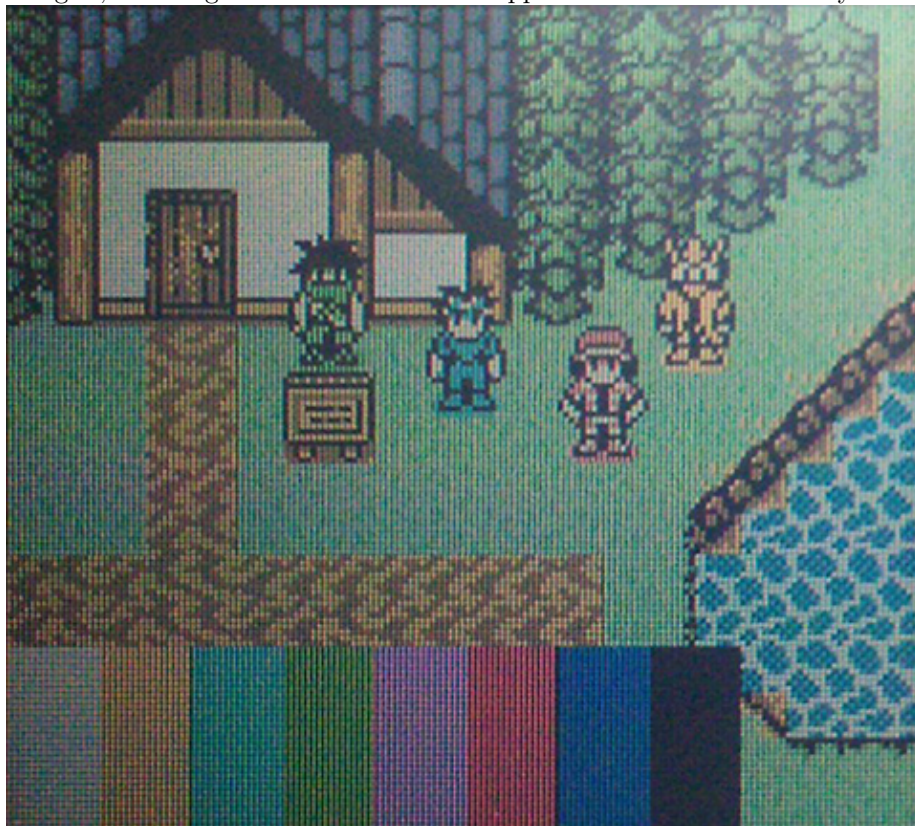
The Game Boy Color screen seen up close. However, Nintendo handhelds have different types of screen, which affects how colors appear on each screen. This is something to keep in mind when developing games or software. You need to take into account how the user will see the final image as it might not be what you want. I made a scene to illustrate this better, here's how the image looks with raw, unmodified colors.



A GBC scene displaying true colors, as seen without the effects of filters or changes due to output through any display (such as an LCD screen). This scene contains many useful bits to identify differences, like 100% SMPTE color bars (The *Society of Motion Picture and Television Engineers* standard color test pattern), dithering, and colors like brown and purple to help see tones. This is how BGB and GB Studio handle it respectively:



Left: BGB (using default settings) Right: GB Studio As you can see, the colors are not as intense (highly saturated) as on the raw image. This is because both the BGB emulator and GB Studio use a different way to generate colors to emulate how it would look on an actual GBC screen. Other emulators such as Emulicious or SameBoy also provide options for adjusting the color and brightness for the player. The thing is, knowing what that screen is supposed to look like is really hard:



A Game Boy Color screen running the sample scene under cold, white fluorescent lights. How games will look on the non-lit, imperfect Game Boy Color screen depends on many factors, like lighting, for example. You won't get the same tint if you play under the sun on a cool, bright morning as apposed to in a lit room with warm lights. Also, as the previous screenshots suggest, it doesn't use perfect RGB, so colors like the green might have a small hint of cyan, for example. Nintendo wanted to make a cheap, cost-effective screen. This is why it doesn't look very good. In fact, this was Nintendo's trend for many years, to make fun out of limited or cheap hardware. We can see this in many consoles, especially pocket devices. Even up to the *Nintendo DS*, which was under-powered compared to the *PSP* (which also had dull colors, at least the phat model). But this is just the CGB model, the Advance line

of Game Boys is a bit different. This is how the same scene looks on a GBA:



Sample scene as seen on a GBA screen. The colors are a little bit more accurate than on the GBC, but there's still a problem (and it's worse too), the brightness. The Game Boy Advance line, or at least the non-backlit models, suffer from a dark screen. This is why many GBA games looked more washed out than similar games on the GBC. They had to compensate for the much darker screen. Games like Pokémon Gold and Silver can look really dark at night or in caves, even with the light on. The Game Boy Color screen looked dark because of the lack of a light, the Game Boy Advance screen looked dark because that darkness was part of it.



A night scene from Pokémon Gold as seen on a GBA SP AGS-001. However, games made specifically for the GBA were designed with the relatively dark screen in mind, and so the developers increased the value and saturation of colors to com-



pensate.

Mariokart Super Circuit as seen on a GBA SP AGS-001. For the Game Boy Advance SP (AGS-001), they finally added a very welcome addition, a (front) light! Nintendo was almost finally catching up to the technology of the Sega Game Gear and the Atari Lynx! This helped for native games, but not so much for older GBC titles as they still looked dark. Thankfully, the AGS-101 model added a proper backlight, which made the colors pop, almost like true RGB!



Fun fact: Instead of the usual RGB pattern the other screens used, the GBA SP used BGR. There isn't much documentation on this matter online, and it's a shame. We now have bright, crystal-clear IPS or OLED panels which can do near-perfect colors, yet the original look is often ignored. There has been a resurgence for CRT screens (there's an entire subreddit, if you're curious), since many people are now looking for that old-school look that gamers back in the 80's and 90's enjoyed. However, there's little-to-no talk about handhelds, and I think it should be brought up more! Considering the amount of mods that change the original screens of these consoles to brighter, more colorful alternatives, and the amount of people that are interested in visual accuracy.



The sample scene as seen on a modded GBC. As I mentioned previously, these kinds of mods often don't take into account color accuracy. I hope this article helps spark some discussion about color and the differences on not just the Game Boy line, but also other consoles, and why accuracy can sometimes be important. To explore the differences in color display on various emulators and hardware yourself, you can download the sample scene shown in this article here. Huge shout-outs to Emi for providing the equipment and some of the pics, Gumpy Function for helping and Tyler Barnes for the macro shot! SōdoDev

SōdoDev

a/k/a "Anima" – Game Developer Wannabe, Average Internet Enjoyer, Human Being. (he/him) ### Share this:

- [Click to share on Bluesky \(Opens in new window\) Bluesky](#)
- [Click to share on X \(Opens in new window\) X](#)
- [Click to share on Facebook \(Opens in new window\) Facebook](#)
- [Click to share on Mastodon \(Opens in new window\) Mastodon](#)
- [Click to share on LinkedIn \(Opens in new window\) LinkedIn](#)
- [Click to share on Reddit \(Opens in new window\) Reddit ### Related](#)

Creating Multiple Dialog Frames using UI Palette Tricks

GB Studio offers lots of options for customizing your dialog frames, but one thing that you can't do is have more than one. There is however a method of swapping the UI palettes to create different dialog frames using a single image. This can allow for simulating different interfaces like a computer terminal, an instant message, or even reading a book. It should be noted that because this method uses color palettes to switch the dialog frames, the effect won't be viewable with any non-color Game Boys (though it is possible to check if the player is on a color compatible device with an **[If Device Supports Color]** event). When you create a new game in GB Studio it will automatically create an 'assets' folder. Inside this folder, the assets are split into four categories. In the 'UI' folder you will find the assets used to create the frame, cursor, and all of the letters in your game. The 'frame.png' file is made of $9\ 8\times 8$ pixel squares representing the edges and interior of your frame. The final frame is created by repeating the top, bottom and sides across the whole screen. You can edit this file to create any frame you want, but it must retain the original four colors. In GB Studio the color is added while building your game, as opposed to editing the original image file. The key to creating frames that look different is to reuse the same color in multiple palette selections for an image. This allows you to turn elements of the design on and off. Here is an example: In the lower frame, you can see that colors 2, 3, and 4 are all the same in its palette. As a result, a lot of extra information is hidden that can be activated by changing the colors again. Once you have created the frame you want, you will need to add the palettes you want to use to your game. You can press **Ctrl+6** or go to **View > Palettes** to get to your game's palettes. You can click the **plus [****+****]** button to add a new palette. I always include the letters 'UI' in the name of my UI palettes so that they will be faster to search for when creating my game. This screen is pretty self-explanatory. You can paste the hex value of a color into the text field on the bottom to import a color from your image editor of choice. Keep in mind these colors will replace the four shades of green in your original image file. Even though it says 'lightest' and 'darkest' you can invert the lightness to create more variety in your dialog boxes. Here are a couple of examples of what you can do with this method if you use some creative problem solving: Remember these colors also affect the text and the character avatars. When creating an avatar, you will need to plan which palette will be used when it is displayed. You can even change the font to be wider in some palettes than others by editing your 'ascii.png' file! Now that you have your frame created, you will need to implement changing it when you need to in your game's code. You do this with a **[Palette: Set UI Palette]** event. All you have to do is search for the palette you created by name. Keep in mind that this will permanently change the palette color, so if you only want it to only change it for this dialog box, you will need to set it back to the default palette after the dialog is displayed. I hope you found this tutorial useful! I'm looking

forward to seeing some cool UIs using this method. Ben Jeltron

Ben Jeltron

Independent game designer, comic artist and art instructor. Creator of *Opossum Country*, *Heliosphere*, and many other comics and games including the upcoming Game Boy game *The Machine*. (he/him)

benjeltron.itch.io ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### Related

Development Workflow – Chapter 9: Creating a Color Palette (Part 1)

A while back, I covered color theory in the context of game design. We explored how palettes can be applied to backgrounds and sprites in order to direct the player’s attention toward points of interest, liven up a scene with analogous colors, and improve readability overall. For the beginner, I suggested simplifying matters using 1-bit or minimal palettes, or by incorporating a pre-made palette into your own game scene. This article is for those who wish to go beyond the basics of color theory. We’ll explore how to pinpoint a color you want, how to expand on that color to create a palette, and then organize that palette into a useful color ramp which can then be applied to any scene to make a cohesive whole. Before reading on, feel free to take this moment to read my chapter on color theory and refresh yourself on the various color theory terms and definitions that will be used in this article. # Picking A Color Pick a color, any color. It’s simple, right? Well…of course it is! We just need to know what we are coloring and we’re away. For the purposes of this article, I have drawn a simple tree using the default colors of GB Studio’s .png template.

A tree, the art element we will be painting in this article. The first color that comes to mind when I think of a tree is green. But even something as simple as determining the colors of a tree’s leaves can be a loaded question. What if you want to set your game in a season other than spring? Autumn or summer may have you wishing for a rich red or perhaps a dry yellow color. Or what if you want to provide a certain mood or tone that is specific to your game? Maybe you’re making a horror game or some gritty dystopian adventure. Then you may be thinking a lush green color isn’t the best solution. Maybe a dark blue or purple would be better? Or perhaps a green hue shifted toward orange or brown? The color you assign to any element *in-game* will depend on a number

of factors. It's up to you to determine what colors will best suit whatever mood your game needs, but we will stick with good old green here. Something vibrant and inviting.

The first pick, a mid tone green. The most direct and simplistic way to extrapolate a palette from this color is to shift its *value* in both directions until we end up with a light tone, two varying mid tones, and a dark tone of the same *hue*.

The same green hue, but with four varying tones achieved by shifting the value parameter. What we have just made is a *color ramp* made of four colors, each of which we refer to as a *step*. Here it is applied to the .png file:

Our four-step color ramp applied to the tree GFX. # **Extrapolating Your Steps** Shifting the value has certainly given a sense of depth, but it's not the most thrilling collection of colors. In fact, it's not too dissimilar to the original .png we started with! Luckily, value is only one parameter at our disposal when coloring an element. To illustrate just how dynamic an image can get through palette selection alone, we're going to move out of the realm of pixels and into the world of fine art for a moment. Let's take a look at a painting by the impressionist John Singer Sargent to show you exactly what I mean.

Villa di Marlia: Lucca, John Singer Sargent 1910 I've selected this piece for its striking contrast between highlight and shadow. You can see that the grass is shifted towards varying yellow hues when in direct sunlight while areas of grass in shadow have been heavily shifted in the opposite direction, towards blue. The stone masonry detail and red dirt path have been painted in the same way, mixing to create shades of purple. The end result is a vivid and exciting exploration of light and form.

A color ramp of colors that give the illusion of grass on paper from Villa di Marlia: Lucca. The highlight is a strong yellow but shifts towards a blue hue as the tone gets darker. But how does all this relate to selecting a palette for pixel art, I hear you say? Well, whether creating a digital pixel art tree or using analogue methods, such as with oils, watercolor, or any medium for that matter, hue, saturation and tone are all there to help create the illusion of form, or how light bounces off the surface or transmits through matter. So the same techniques and rules that an oil painter uses to create a palette before laying brush to canvas can be used just as effectively when a GB dev creates several palettes before painting any art element with some form to it. So with that, I'd like to share some rules that I use when not only creating a color ramp to paint a scene in a game but also when mixing my own oil paint palettes. # **The Rules of Color Shifting** If I had a mid tone red oil paint and wanted to create a highlight from this color, I would choose one of two colored paints to mix in. Firstly, I could choose yellow paint. It brightens the value of the red, and also shifts the hue towards a warm orange color. Or I could use a white paint. This would also increase the red's value but, this time, I would be shifting its hue towards a more cool color, in other words, blue. Same again for if I wanted to make the red color darker. I could pick a dark blue, mix it in, and watch the

red paint turn to purple, slowly adding more until I get a darker purple and so on until I come to the desired colored paint. I would end up with some variation of a dark shade of cool purple. Or, I could add brown (which is a dark orange color if you want to pinpoint that on a color wheel) and I would end up with a darker warm red shifted towards yellow. In the context of digital art, shifting hues along a spectrum is akin to mixing in these yellows or whites, or these dark blues or browns into a base colored paint – only you are sliding a toggle on screen rather than physically mixing paint together, obviously. In any case, the same rules apply.

An example of the toggles you could move when shifting from a red to a lighter color in GBS' s Palettes view. In the above image I shifted the red mid tone color in several ways to arrive at a lighter yellow highlight color. Moving the brightness (value) toggle to the right lightens the color; that' s simple enough. I also started fiddling with the *green* and *blue* toggles by shifting them to the right and left respectively. This in turn shifts the hue to the right and towards the yellow area of the hue spectrum. In this way, you can nudge any color you want in any direction you want to suit your needs!

Here are the rules in a handy table Let' s return to our tree and redo the same four color ramp, this time utilizing the rules above:

The color ramp and tree GFX using colors shifted only with value (above) while the new color ramp shifts both value and hue (below) This new palette is a bit more exciting than the original, and I could easily push the hues further apart to make an even more dynamic four-step palette. Experimentation is the name of the game here, try nudging colors in various directions, play with parameters, find out what works and what doesn' t. Even though it seems like palette creation is some magical mystery for the beginner at times (and even for the intermediate), it really is just a matter of practice once you know the rules. Having said that, we aren' t done yet! In Part 2 of this article, we will expand our four-color palette into a versatile *color ramp* with multiple colors, branching paths and color bridges. By the end, you will have the knowledge necessary to create a master palette that can color an entire game *and* ensure the game' s palette is varied while still remaining cohesive as a whole.

This article is part of a series: Development Workflow Gumpy Function

Gumpy Function

Independent Games Designer, Artist, Film Enthusiast and Full-time Dad (he/him). Check out my games here!

www.gumpyfunction.com/ ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn

- [Click to share on Reddit \(Opens in new window\)](#) [Reddit ### Related](#)
-

Development Workflow – Chapter 9: Creating a Color Palette (Part 2)

From Steps to Color Ramp

In Part 1 of this article, we learned how to create a step ramp from one color to another. Let's keep expanding our four steps into an expansive color ramp that incorporates multiple colors. This way, we can include some reds and browns and assign them to the tree's trunk and the flowers.

Our initial four-step palette and the Tree GFX we will continue painting Creating a *color ramp* can take a lot of work. Shifting colors around takes time and there is often a considerable amount of further nudging once you start testing out the ramp and realize some colors just aren't working well enough to support others. But it's all worth it! Because once you have a master palette in the form of an interconnected color ramp, it acts as an easy to use guide when creating any number of palettes within GB Studio for your own game. To start, let's pick one of the four colors, and start extrapolating from there. We could pick the dark green-blue color and shift towards a lighter red, passing through various blues and purples, for example. Or we could start from the yellow and shift through to orange and then red and so on. When building your own, you can do whatever feels right for you but for now, let's start stepping to red from our yellow color.

Branching the color ramp out from the yellow color towards a darker red I have used the same rules described in Part 1, shifting the value to a darker tone as well as shifting the hue towards red as I go, *being careful not to create too small or large a step*. Now it's a simple matter of repeating the process, picking an existing light color and shifting it towards a new dark color or vice versa.

A more complete color ramp with a wide range of colors Now we are getting somewhere! This color ramp features a wide range of hues and all the steps are designed to shift gradually from one to the next. You can see that I could select any four linked steps here and insert them into a GBS palette to paint a tile or a sprite in-game with ease. Notice there are three branching ramps, one for blue colors and two gray ramps (one warm and one cool). Our yellow to red ramp is now a *color bridge*, and there is a large *color loop* making a circle across the ramp. We have just made a huge leap in complexity here, so I'll take this moment to break down what I did in a little more detail. # Using Image Editing Software Outside of GBS Personally, I find it easier to make a ramp outside of GBS using GIMP (or a similar image editing software). The *color picker tool* in an image editor is not only a relatively quick and painless way to nudge colors around but also visualize how far away each step is in terms of hue, value, or

saturation. Here are some snapshots of some of the steps within our color ramp as an example:

The four blue steps in one of the branching ramps as seen in GIMP's color changer tool. In the above example, I've shifted the dark purple towards a cool light blue by doing the following: * Increasing the value upwards, making the color more bright. * Decreasing the saturation slightly, making the color more "white". * Shifting the hue down away from red and towards blue. Take a moment to study the image below to see how the three points above can be translated visually using the color picker of your chosen image editing software:

The direction of movement through hue, value, and saturation as seen on GIMP's color changer tool. As you can see, our four-color blue ramp is following the red directional arrows as indicated above. If we were to measure the difference between values from step to step, you will find that each is around the same. This will ensure each color steps up or down uniformly across the ramp. The same is true for the hue and saturation. In essence, you are looking to draw a straight line from the darkest color to the lightest color as you shift from light to dark and through the hue spectrum. How complex and interconnected a matrix of colors you want to create is entirely up to you. This color ramp may seem like quite a leap forward in complexity (and to be honest, it is), but the process is exactly the same as our initial four-step ramp from yellow to a dark green. The more you practice, the more comfortable you will become expanding a palette's complexity. Now it's time to test our colors by painting our tree GFX. In this way, we can figure out if any of our selected colors aren't quite working to support the overall palette.

The tree GFX colored using some of the steps from our color ramp. I've picked a few linked steps to color the trunk in brown and the flowers using some pink and orange colors. If we draw our attention to the pink flowers though, I think we could work on improving that strong pink color of the petals so that the flower isn't "jumping out" at us so much. I've purposefully made this pink all together too intense compared to the more muted highlight of the flower's top surface to illustrate an important point when creating a cohesive range of colors. And that is; manage your *saturation* or *the level of intensity of colors*. # Saturation Dealing with changes in saturation levels across a palette is perhaps the most difficult parameter to get right, at least in my opinion. Having said that, there is nothing wrong with changing the saturation of colors across a palette, in fact it's necessary to mute various colors and arrive at much needed gray scales or whiter variants of a color. At the same time, saturation contrasts can also be incredibly nuanced and difficult to get right! Let's return to the work of John Singer Sargent just to show you how effective creating a high saturation contrast between elements in a piece of art can be.

John Singer Sargent, *A Dinner Table at Night*, 1884 In *A Dinner Table at Night*, the vivid reds of the lamp shades and surrounding light are used to direct the viewer's attention to the female figure in the center of the painting. The rest of the colors are muted variations of blacks, grays, and yellows relative to the

red. It is the contrast in color saturation that makes this piece such an effective use of color. In the context of game design, saturation contrasts can be used not only to create interesting static screens for cutscenes or menus, but also as a way to differentiate art elements to support the gameplay itself. For example, using muted colors can push background art further into the background, while using saturated colors will pull an art element into the foreground. As seen in the screenshot below, this can be used to improve readability and create a sense of receding dimension to a game's scene.

Samurai Kid, Biox (2001) As you can see, creating a strong saturation contrast within a scene can lead to great results. It can be a highly effective weapon in your game design arsenal, but if you are unsure about how to deal with saturation levels across a palette and want to keep things relatively simple as you learn, then I offer this advice to you: *Stick with a similar amount of saturation across a color ramp*. So if your game is designed to appeal to kids (or the kid in all of us), with the exception of shifting towards grays, keep the saturation of your color ramp relatively high. If you are going for a more relaxed or adult vibe, then you would ensure your color ramp features some degree of muted colors to support the game's premise. To reiterate, if you are ever unsure, as long as you don't start shifting the saturation's of mid and dark tones all over the place, your palette will work out just fine. With this in mind, it's time to nudge some colors in the palette. In particular we need to make sure the step between our flower highlight and the pink petal isn't too large of a tonal shift both in value and saturation.

The pink color has been desaturated considerably and the purple has also been slightly desaturated. Now they blend more easily with the saturation level of the grass and don't "jump out" as much. We would then make the necessary edits to our master color ramp and continue tweaking and fine tuning it by applying colors to the scenes and art elements in our game, learning more about our palette and its strengths and weaknesses as we go. Now that we have improved our color palette ramp, we can create the necessary palettes in GBS and paint our scene.

A final touch-up with a .png prepared for coloring and the necessary palettes with a more uniform saturation created in GBS Picking a palette, creating a color ramp and fine tuning it can take time and quite a lot of patience. But it can also be a fun way to explore color as well as provide a base for you to come to when coloring your entire game in an effective and cohesive way. If you've never tried creating your own color ramp, I hope this article has helped to get you started. Yes, making a palette can seem like a perplexing problem but remember, start small, take your time, experiment and find what works for your own game. With a little practice, you will be creating not only aesthetically pleasing palettes but also palettes that can influence the very experience of the player in a substantial way, and your game will be all the better for it!

This article is part of a series: Development Workflow Gumpy Function

Gumpy Function

Independent Games Designer, Artist, Film Enthusiast and Full-time Dad (he/him). Check out my games here!

www.gumpyfunction.com/ ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### *Related*

Lights Out - DMG Custom Color Palettes Tutorial

Editor’ s Note: This article contains flashing images for those who are sensitive to them.

*This tutorial is built heavily on a plug-in created by **NalaFala (Yousurname)** and would not be as easy to implement without its help.*

We will be focusing exclusively on the DMG Palette for this tutorial.

With the horror themed homebrew games filling a void in the Game Boy’ s library, let’ s explore and implement a classic horror trope: lights out. Deliberate manipulation of the light and shadow can be used to heighten tension, obscure paths, cloak sinister beings lurking in the darkness, or even the tried and true jump scare. ## Source Code <https://github.com/phinioxGlade/gbstudio-dmg-custom-color-palettes> ## Playable In-Browser, or Download the ROM <https://phinioxglade.itch.io/gbstudio-3-custom-dmg-color-palette-tutorial> ## Key Concepts * Using the set palette colors plugin * How DMG color palettes work * Setting the color palette of background layer * Setting the color palette of sprite layer * Arthoring art to work to leverage non standard palette * Color palette cycling * Custom fade in and out effect # Set Palette Colors Plug-in You’ re probably already aware of the built-in events for [Set Background Palettes] and [Set Sprite Palettes], unfortunately these only apply to GBC palettes, but luckily a plugin made by NalaFala (Yousurname) called Set Palette Colors is here to help those making DMG games. An image of the Set BGP Colors event in GB Studio. # DMG Graphics Refresher The Game Boy has a 2-bit color palette which can render sprite, background and overlay window layers. The sprite layer (OBJ0) supports 3 color channels with an additional transparent channel. The background and overlay window are both tile layers (BGP) and can be either 4 color channels, or 3 color channels plus 1 transparent channel. The BGP transparent channel only applies to sprites that “display behind background layer” are checked in the sprite editor.

The color palette for the sprite and background are independent but apply to the entire layer. The [Set Palette Colors] plugin allows us to reassign the color that each channel is using, i.e. swap white to black and vice versa. A set of screenshots, showing palette cycling on a background in GB Studio. ##

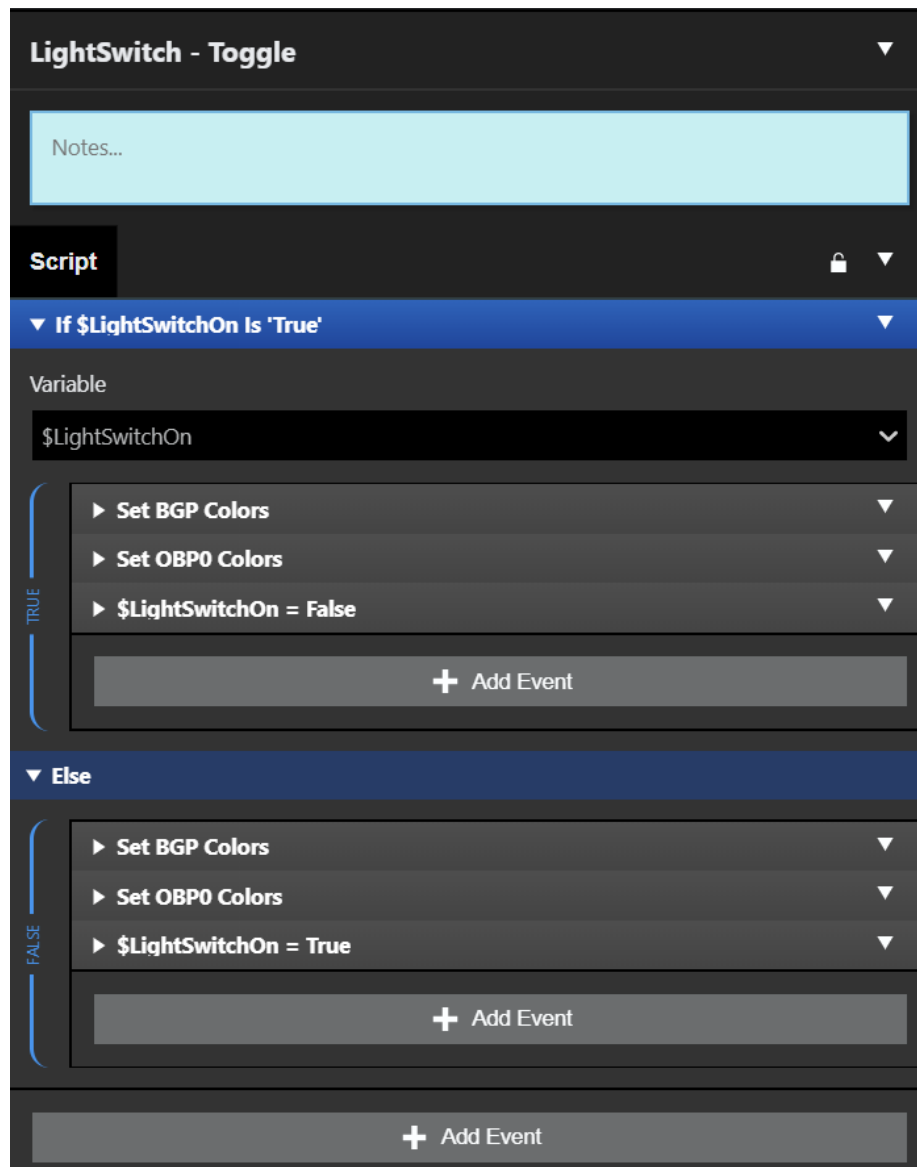
1. Swapping Black for White We will start with the simplest demonstration, setting a single background color channel from black to white: 1. Create a new project called LightsOut using the built-in Sample Project (GBS V3+) you will be using these assets. 2. Install the setPaletteColorsPlugin: 1. Close GB Studio. 2. Navigate to your project's root folder and add a new folder called plugins. 3. Inside the plugins folder add a new folder called setPaletteColorsPlugin. 4. Inside setPaletteColorsPlugin copy of the contents <https://github.com/YOUR-USERNAME/gbs-plugin-collection/tree/main/plugins/setPaletteColorsPlugin>. 5. Reopen GB Studio and the project. 3. Delete all the scenes. 4. Add a new empty scene called "1. Black for white on INIT" and select the black background. 5. Add set palette color event to Scene OnInit and set Color 4 to a value of 0. 6. Run and you will see the background is white instead of black. A set of screenshots, showing a scene in the GB Studio tool versus how it is displayed in game after the event is processed! ##

2. Light Switch Let's build a simple light switch. The Player will interact with the switch, allowing them to toggle the room's lights on and off: An animated scene in GB Studio. The player character interacts with a device, which changes the lights in the scene. 1. Add a new scene called "2. Light Switch" . 2. Set the background as the house. 3. Create a global variable called LightSwitchOn. 4. Add a [Set Variable To True] for the LightSwitchOn variable on Scene On Init. 5. Add a new sprite, rename it to LightSwitch and set to the Machine sprite sheet. 6. Add a trigger over and in front of the LightSwitch sprite. 7. Add "Attach Button Script" event to the LightSwitch's OnEnter. Inside the Button script code needed to toggle the lights: 1. Add If variable is true to evaluate LightSwitchOn: 1. Inside If True: 1. Add "Set Palette Color" event with the values 3,2,3,3, this will most of the room black. 2. Set LightSwitchOn variable to False. 2. Inside Else: 1. Add "Set Palette Color" event with the values 0,1,2,3, this will restore the palette to its default. 2. Set LightSwitchOn variable to True. A screenshot from GB Studio, showing the events required to create a scene. ##

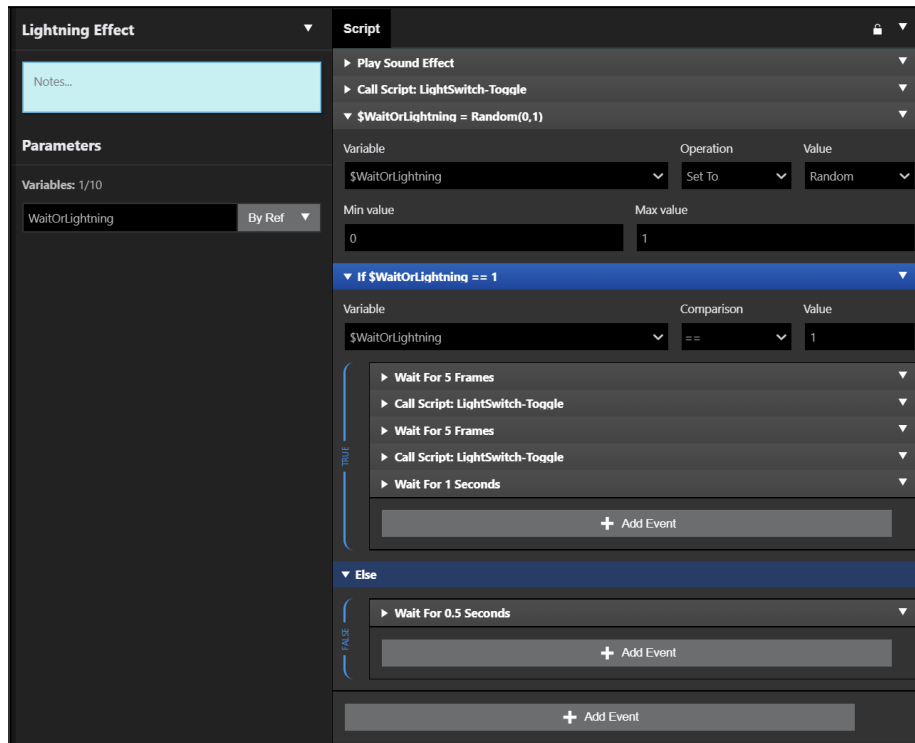
3. Light Switch – Include Sprites Now that we have the lights affecting the background, the sprites look out of place in the darkness. To solve this, we are going to set the sprite palette: #### 1. Duplicate the "2. Light Switch" scene and rename the new scene "3. Light Switch – Include Sprites" . 2. In the "Attach Button Script" If True clause: 1. Add "Set Palette Color" event, set to OBP0 and the Color channels to [2, 2, 3] – this makes the sprites dark but not invisible.



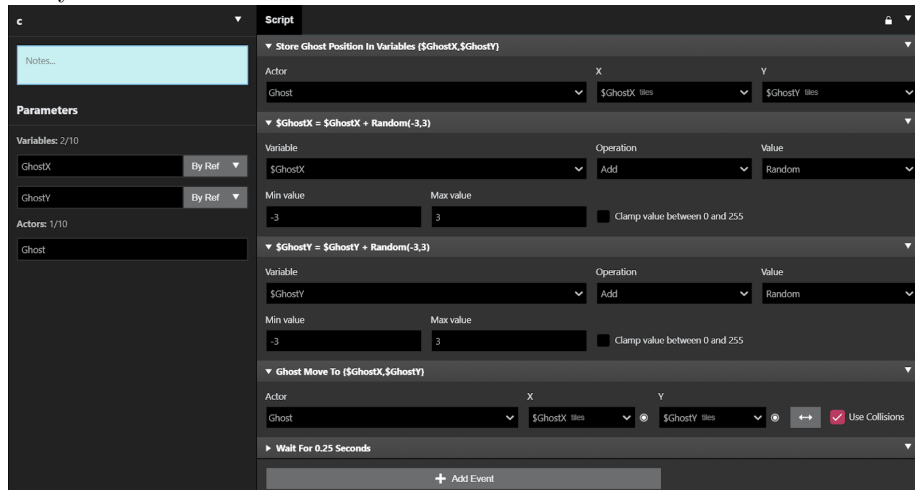
3. In the “Attach Button Script” If False clause: 1. Add “Set Palette Color” event, set to OBP0 and the Color channels to [0, 1, 2]. 2. This restores the sprites to the default palette. ## 4. Lightning and Ghosts We will finish this simple tutorial by implementing a gameplay gimmick, randomly lightning will illuminate the room revealing ghastly ghosts floating around the room: An animated scene from a GB studio game. The lights in a room flicker randomly, displaying enemies visible only when the lights are on. During the darkness only the ghost’ s eyes will be visible. This will require the creation of a sprite sheet that takes into account how we are altering the color channels. A screenshot of GB Studio’ s sprite editor, showing the frames used for the actor to achieve the invisibility effect. 1. Create a new scene: 1. Set the background to house. 2. Rename the to “4. Lightning and Ghosts” . 3. Scene On Init set LightSwitchOn to False. 2. Create a new global script called “LightSwitch – Toggle” : 1. Copy the contents of the “3. Light Switch – Include Sprites” s “Attach Button Script” If True clause in the into “LightSwitch – Toggle” .



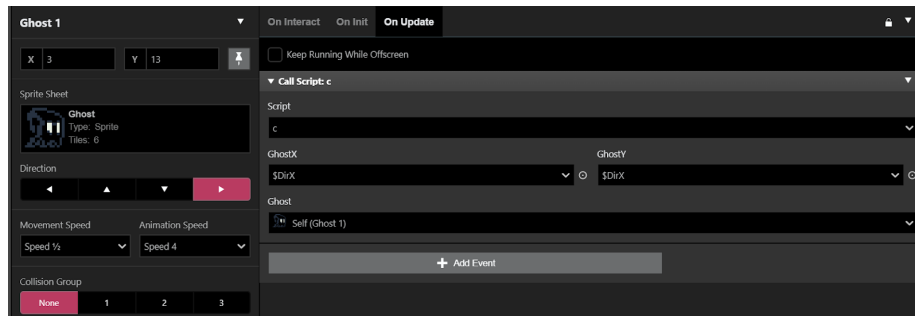
3. Create a new global script called “Lightning Effect” .



4. Create a new global script called “Ghost Patrol” . Randomly calculates how many tiles it will move and what direction.



5. Build the Ghost sprite with its animation. 6. Add a new actor to the scene:
 1. Set the sprite sheet to Ghost. 2. Rename “Ghost 1” . 3. On Update call the Ghost Patrol global script.



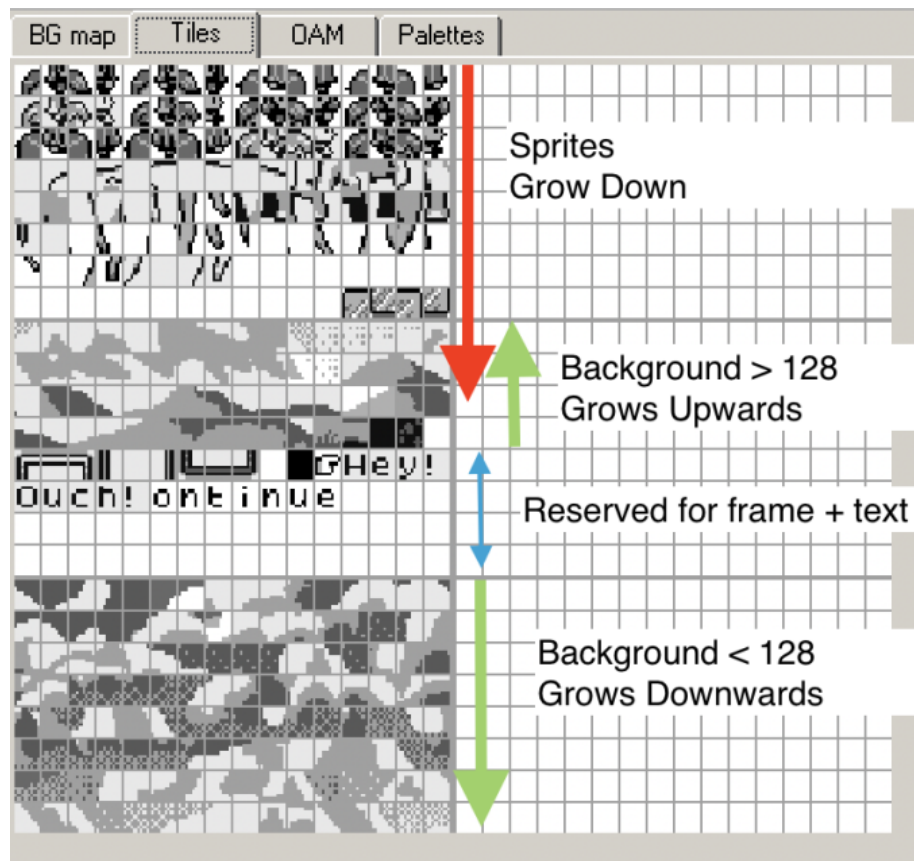
7. Duplicate the Ghost 1 and rename Ghost 2. *# Effects* To close the tutorial we will demonstrate some of the effects you can achieve. *## Cycling Color Channels* By cycling the colors you can recreate the star powerup effect from Super Mario. `|||||***`

Managing Sprite Tiles

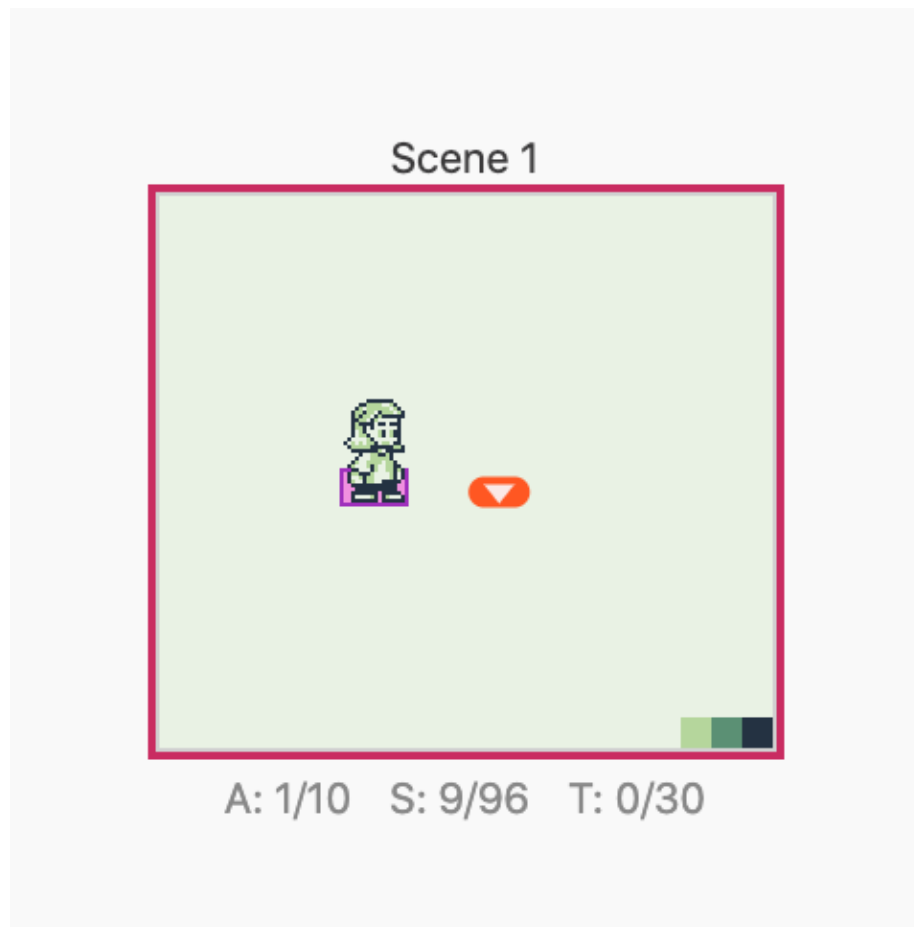
In my previous *Sprite Editor* article, I went over how to use the *Sprite Editor* in GB Studio to create your own sprites and animations. After playing around with it a bit, you have hopefully grasped the main concepts and are comfortable with it. There are a few nuances in terms of how sprites work that makes it beneficial to examine in depth, so let's take a deeper dive into how GB Studio handles sprites and how you can get the most out of it. *Note: this article assumes you already have an basic understanding of the Sprite Editor, so make sure to read the previous article if you are unsure of anything.* *## Limitations* The reason why you'd want to learn about tile management is because even though GB Studio 3.0 opened up a lot in terms of resources – they are still limited. This is partially due to how GB Studio handles sprites, graphics and memory, and also just the very nature of the Game Boy hardware and how it works. Let's look at the Game World and see what happens when we add an actor to it. For this example, I'll be using assets from the Sample Game but, for simplicity, I will be placing them into a blank project.



A new scene using the placeholder background has three definitions below it, A: 0/10, S: 0/96 and T: 0/30. A stands for *Actors*, S is for *Sprite Tiles* and T are the *Triggers* (you can click on any of these numbers to see that explanation). By using the placeholder as a background, only 4 tiles are being used to build it which means we have 96 tiles left for sprites. If we were to use a more complex scene (more than 128 tiles), this number would decrease to 64. This is because the graphics space in memory is shared between the backgrounds, sprites, text, emotes and UI. The following graphic demonstrates how things are allocated when a game is loaded. As you can see, the sprites and background tiles can “spill” into each other and share the graphics space.



The total number of tiles combined allowed in a scene is 384, so it is possible to exceed these limits if you decide to not use any text, but it then becomes a balancing act of manually taking into account what tiles you have left to work with. With this in mind, it is a good practice to follow GB Studio's recommendations when placing your actors, but we'll expand on the "true" limits in a later article. For now, let's place an actor and see what happens. We'll start with the *player_platform* sprite, and only have the idle animations loaded in the sprite editor.



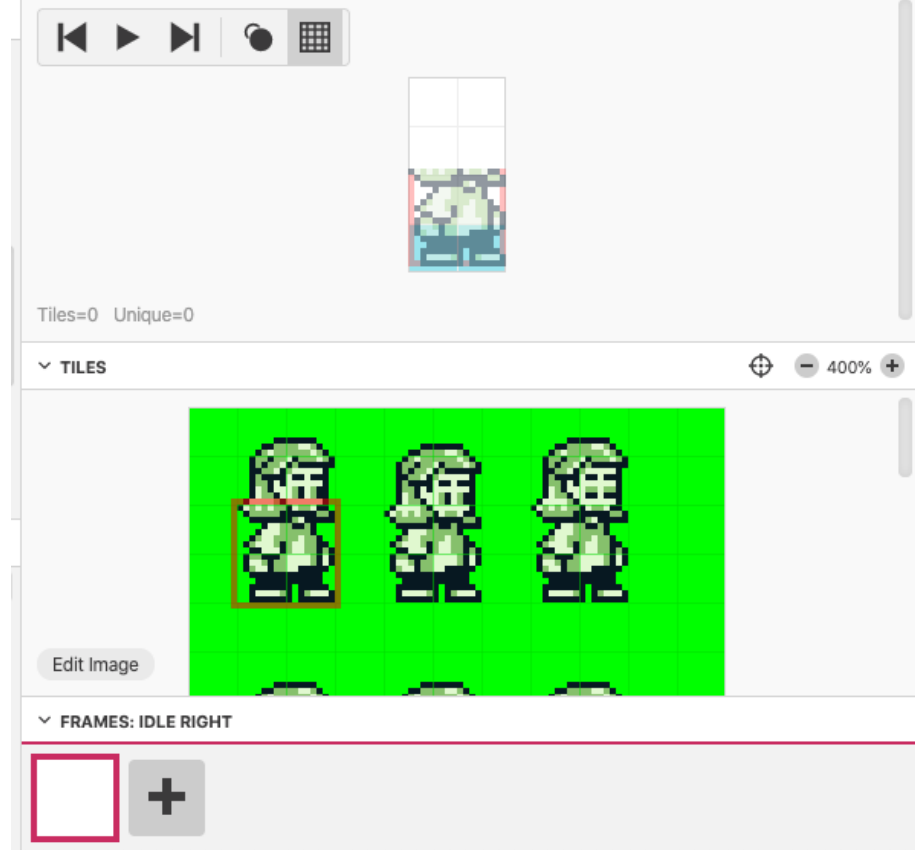
When added to the scene, you can see the total number of *actors* changes to 1/10 (makes sense, we just added 1), and the *sprite tiles* reads 9/96. But where does this number come from? Currently, the *Actor* has an idle animation set to it, which is itself made up of 6 *Animation Frames*, each one 2 sprite tiles wide and 4 sprite tiles tall. So you might think the math would add up to $6 \times 2 \times 4 = 48$ sprite tiles. While the total number of tiles being rendered by the Game Boy to display that sprite are in fact 48 over the course of its animation cycle, many of the tiles within the frames are duplicated. In the same way that a background is built by making a tileset index and laying it out over the scene to save on memory, GB Studio automatically detects duplicate frames and “reuses” them to save on graphics space. This can be demonstrated using Pautomas’ tile counter tool, which shows a heat map of any image loaded, in this case the `player_platform.png` from the sample



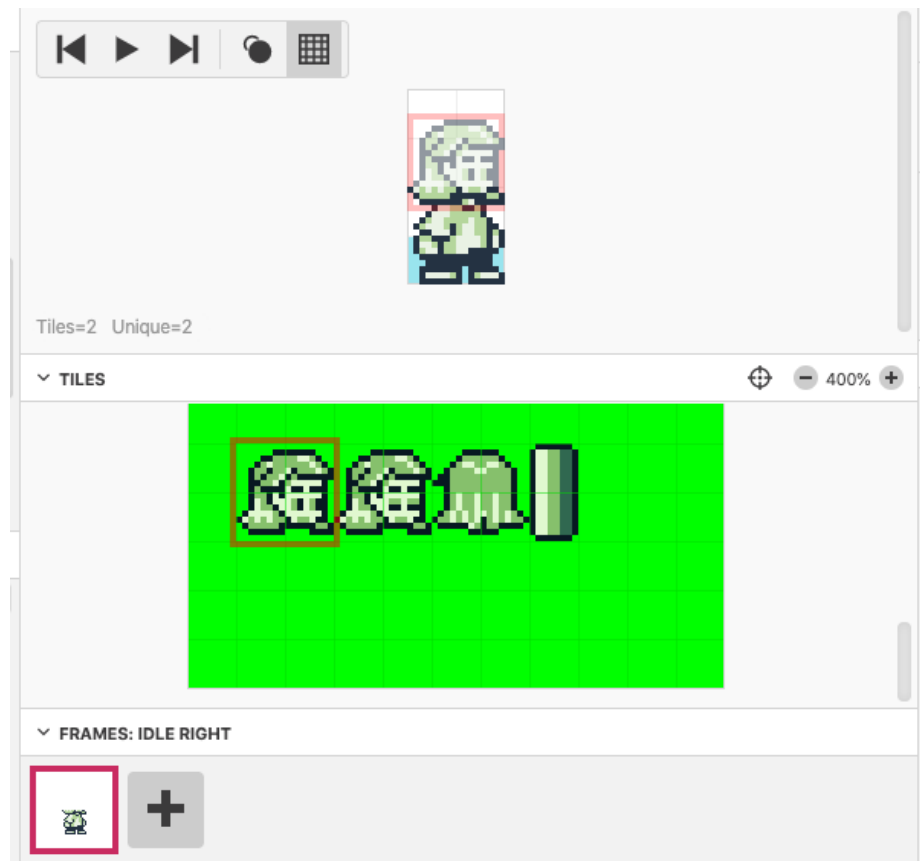
project:

With this tool, the more red a tile is, the less often it is repeated in the scene. This can be used to help developers and artists when they are trying to figure out where they can save on tiles for backgrounds as it will point out potential issues. In addition, it can also help visualize what tiles are being repeated in our sprite sheet. Something like the green background is used over and over again, so it has no red at all, but some of the tiles, like the face and idle animation body parts are also fairly “light” in terms of this heat map. GB Studio performs a similar process when building the sprite data looking for duplicate tiles to optimize for your actors. Even though this looks good from a utilization

standpoint, there is a better way to optimize your sprite tile usage, allowing you for more complex sprites and animations. For the example I used above, I copied the same technique I used from my previous article, where I selected entire sprites as they were laid out in the sprite sheet in the first row, but notice that at the bottom of the sprite sheet there are disembodied heads? When you select a range of tiles from the sprite sheet and place them in the canvas, you are actually placing multiple 8 x 16 sprites together to form a “metasprite”. These do not have to line up with the original sprite sheet components at all and can be repositioned, rearranged and layered to get more flexibility while also saving on your tile counts. Let’s try it out. Make a duplicate of the *player_platform.png* in the **assets>sprites** folder. For the first frame of the idle animation, highlight just the bottom half of the first sprite in the tile selector and place it in the bottom portion of the canvas.

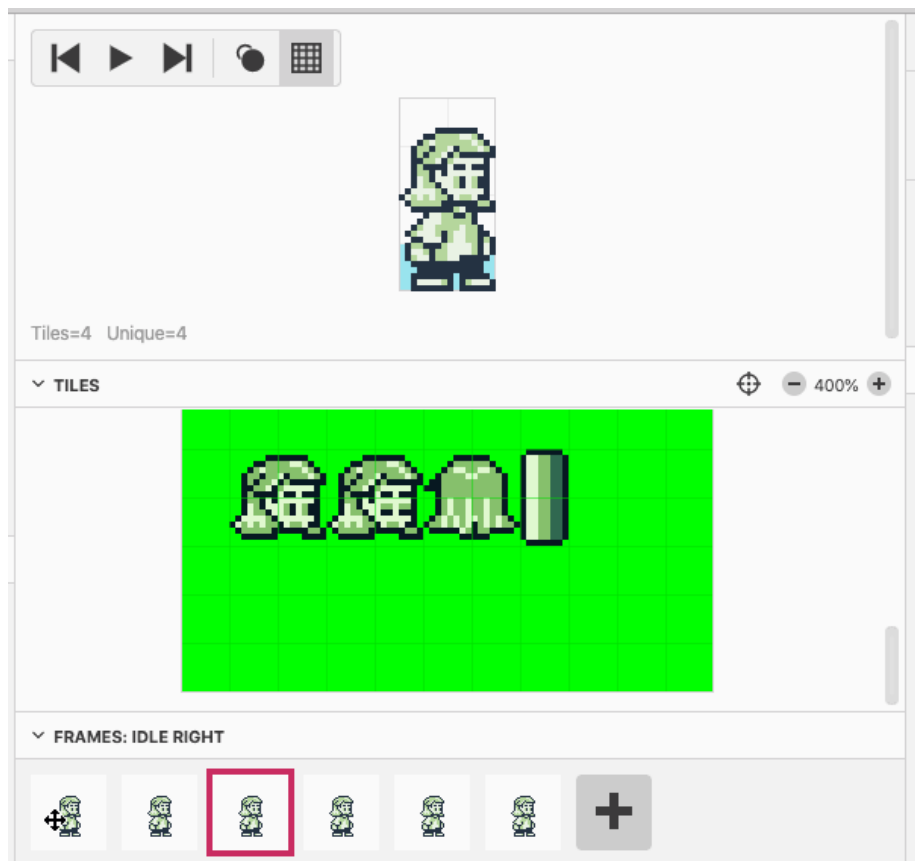


Next, scroll down the sprite sheet in the tile selector and highlight just the head. Place it on the canvas and overlap the body where the hair lines up. If it doesn’t line up on the first try, you can use the arrow keys on your keyboard to nudge it

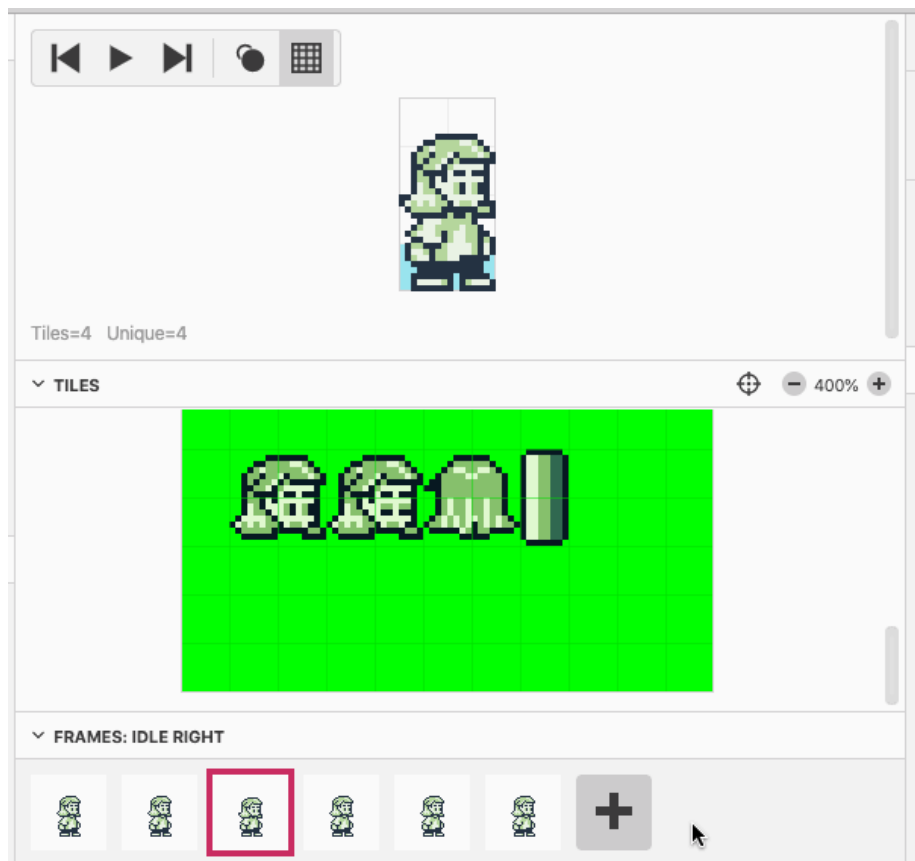


into place.

Copy and paste this frame five times to get six frames total. Now select frame three, and hold shift while clicking on the two sprite tiles that make up the head in the canvas. Nudge these down one pixel to mimic the look of the head bobbing frame from the sprite sheet.



Finally, let's replace the face on the last frame to the version where the sprite blinks in the sprite sheet. Since only the right half of the face is different, you can select just that portion in the canvas, delete it with the delete/backspace key on your keyboard, and then select just the right side of the face in the tile selector and place it.



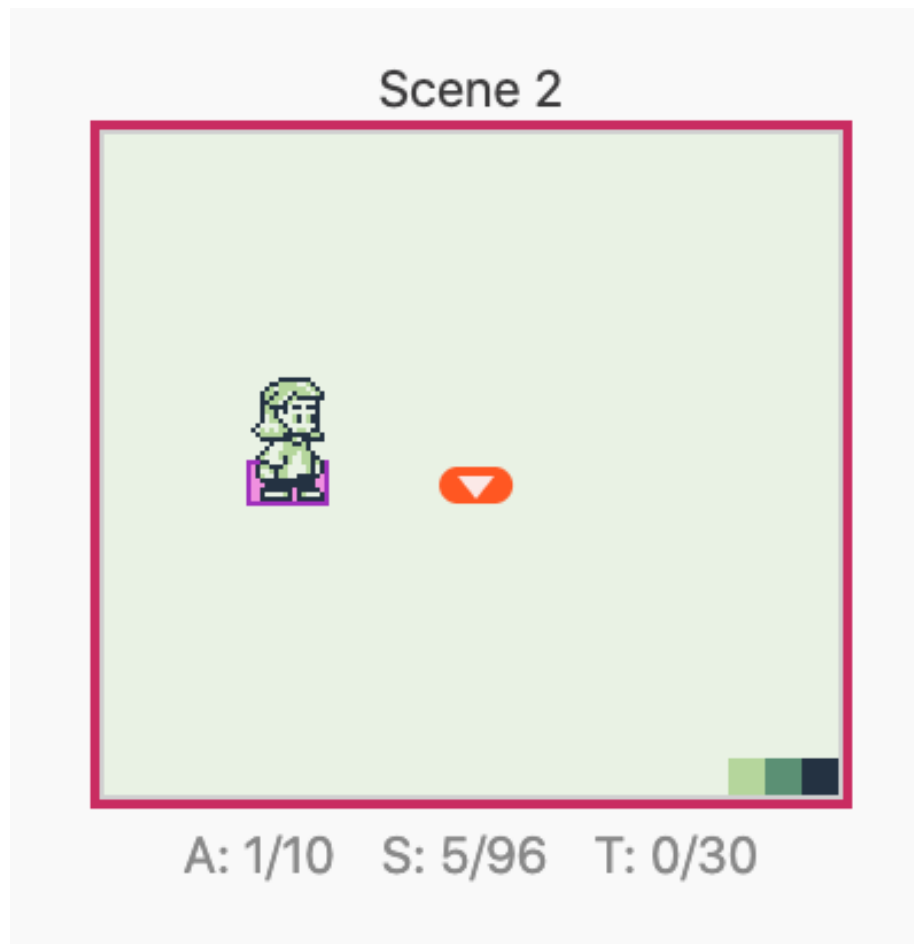
What we have now is an almost identical idle animation to what was created prior, with just a bit of an overlap on the hair giving away that third frame in the “bobbing” animation. With a bit of time, one could edit the sprite sheet so that the body was completely separate to fix this. Here’s the original animation:



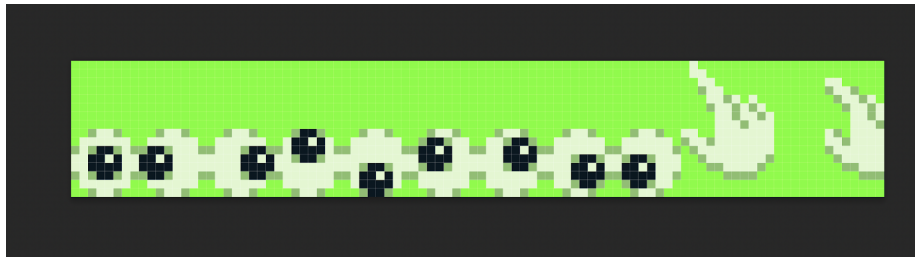


And here's the "optimized" one for comparison.

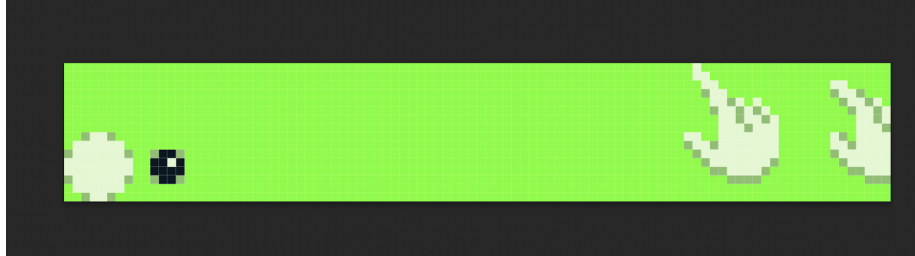
So what was the point of all this? Take a look at the "Unique" count under each animation listed above. The original animation lists 9 unique tiles for that idle sequence, the same number that was shown in the game world when we added the actor to the scene, whereas the optimized version lists 5! Wow, we've saved almost half the sprite tiles by putting together the sprites manually!



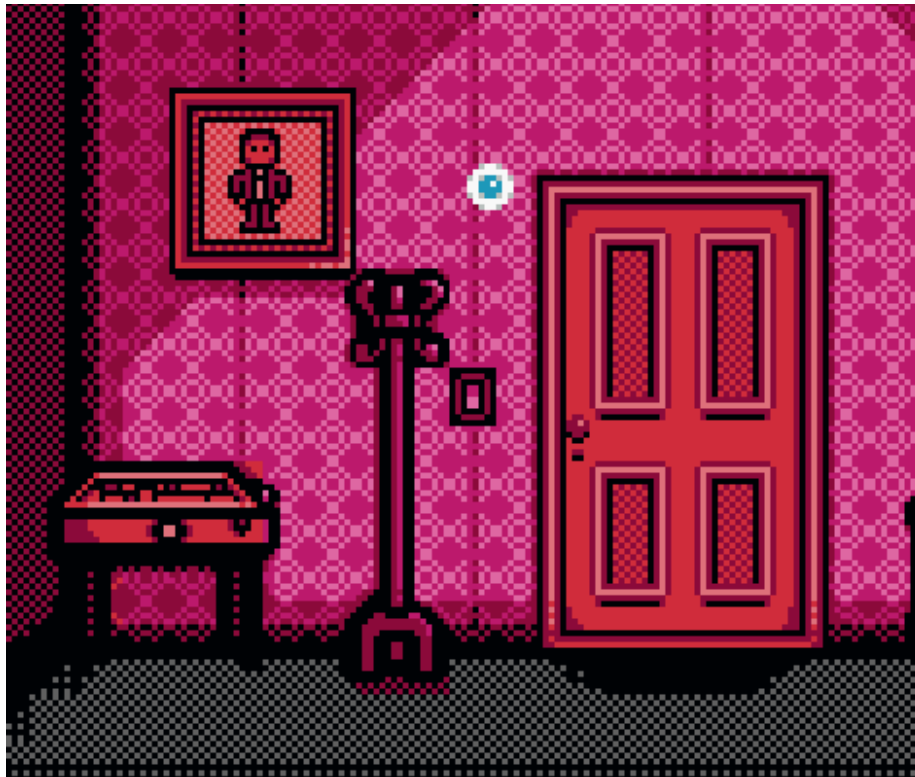
This might not seem like much at first glance, but consider that this was just the idle animation, none of the running, jumping or climbing animations have been placed yet. With a simple background like this, you can get away with “wasting” tiles, but if you want to have more complex backgrounds or more sprites on screen, these overages can add up quickly and you will end up running out sooner than you think. Let’s look at another example. ## 8 Way Movement I’ve been working on a Game Boy port of Kevin Hanley’s excellent NES game, “NEScape!” for the better part of this year. It is a point and click escape room puzzler that has pushed the limits for me personally in what can be done with GB Studio. One such challenge is the cursor. Featuring eight directional movement, the player icon is an eye that “looks” in the direction you are going. I initially thought I could easily cheat this with sprite flipping until I noticed that the pupil has a single pixel denoting a “sparkle” in the eye, and flipping it just didn’t work. In addition to the eye cursor needing all this movement, the icon needed to be able to change according to what the player was interacting with (such as a hand grabbing). Here’s my original sprite sheet to try and solve this:



Since the eye icon is only 8×8 pixels and GB Studio uses 8×16 pixel sprites, this made sense. But wow, that's a lot of frames for just the eye, and when all the frames were placed, this sprite took 12 unique tiles total including the hands. Knowing what techniques are best for making complex sprites like above, can you guess how I optimized this? Here's what I came up with:



That's right. If you look at the eye, the only thing that's really changing is the position of the pupil. So instead of having 9 separate frames for each directional state, I simply stacked the pupil on top of the eye and repositioned it for each directional state, cutting down the sprite tiles used from 12 to 6! This allows for more tiles available for the other sprites I need in the game, which is important as the complex backgrounds don't give me much leeway. The results made me quite happy.



GBscape! uses the same art from NEScape! by peekabrewn That' s it for this part of the Sprite Editor tutorial. This technique of composing complex sprites from smaller components is nothing new. Check out the sprite sheet for *Ninja Gaiden*, notice how the main character Ryu is composed of a “slanted” set of sprites? Games like *The Legend of Zelda: A Link to the Past* also separated Link' s head from his body to give it a little bobble as he ran. Think about how you can separate the parts of your sprite and piece them back together using the editor to optimize the sprites, and don' t forget to use those tile flipping tools too! Some other considerations: * Think about your sprites like paper cut out puppets. What pieces can be reused to save on tiles. * Only tiles used in the canvas will count to the total tiles, so you can create a larger sprite sheet comprised of “full” frames to use as a reference. * The “onion skin” tool is really useful for lining things up. * The size of the canvas does not count towards the tile count, make it larger if you get glitches or artifacts. * Switching animation states is less resource intensive than switching sprites, so try to put all the different sprites you want for an actor in one sheet and use animation states to define them. * You can change the palettes of overlapping sprites to get more colors! Emi Paternostro

Emi Paternostro

Audio Engineer, Mac Technologist and Video Game Developer. Managing Editor

of GBStudio Central. (he/him)

proximitysound.com ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
 - Click to share on X (Opens in new window) X
 - Click to share on Facebook (Opens in new window) Facebook
 - Click to share on Mastodon (Opens in new window) Mastodon
 - Click to share on LinkedIn (Opens in new window) LinkedIn
 - Click to share on Reddit (Opens in new window) Reddit ### Related
-

Sprite Style Differences in Game Boy and Other Consoles

I wanted to talk about something that I thought was kinda interesting, sprite styles; how sprites look and are perceived in their respective games. I think this is a good topic to bring up, since certain GBS games seem to go for sprites without outlines as opposed to using them, like most other Game Boy games. It might not sound deep, but there are a lot of nuances, so let's talk about it. ## Background We could use some background, so let's start with a similar console to the Game Boy: the NES. Released as the Famicom in 1983 in Japan, and as the Nintendo Entertainment System in 1985 in North America, this console was made to be inexpensive but powerful. Inside there's a chip that handles video: the PPU (Picture Processing Unit). For the time it was relatively powerful, but we're talking early 80's hardware so there were limitations. The one we'll focus on for this article is color. The NES graphics could only support 4 colors per tile.

PPU Viewer from the emulator FCEUX. As you can see, all palettes have 4 colors each. Character designs often had the problem of being too colorful, so developers had to find a way to draw them without losing too much detail. Since one color was used for transparency most of the time, there were truly only 3 colors to play with, so programmers tried to use them as much as possible. This is very visible in early NES titles: *Note: the transparent color in the penguins is the same as the bg.* Three colors: for human characters it was usually one skin color, one main color and one secondary color. This worked well for backgrounds with distinct enough colors, and the CRT televisions that were designed for could display them with no problem. However towards the end of the NES era, a new style emerged. This technique favoured one of the colors to use as an outline, which visually left the characters looking like they had two colors: This new sprite style coincided with the release of the original Game Boy in 1989, which featured a monochrome, 4 tone LCD screen. Compared to the CRTs the NES used, this new screen was very different, it had a lot of quirks, which meant it was time to... ## Choose a Sprite Style The screen on the Game Boy was terrible: it was slow, hard to see, and the default color was white instead of the usual black, so games on it had to look a little different to their console counterparts. Also there was only one color, an ugly yellowish green. No fancy

bright distinct colors like the NES. This meant that out of the two sprite styles that appeared on the NES, using outlines made the most sense. The terrible screen forced developers to make graphics as visible as possible, and since there were no colors, the disadvantage of not having a third color wasn't as bad. This became a very popular style amongst Game Boy games released. *Super Mario Land 2, Dragon Warrior I&II and Pokémon Gold/Silver, all examples of games with outlines.* Later, better models of the Game Boy would be released, such as the Pocket, the Light, and the one we'll discuss next, the Game Boy Color. ## The Outlineless Sprites Strike Again! ...sorta The Game Boy Color, released in 1998, featured a color screen with reduced ghosting which afforded much better quality than previous models. This allowed graphics closer to the quality of the NES, with similar limitations too: 4 colors per palette, although you now had 8 palettes for sprites. This allowed developers to...stop using outlines? Not many games did this, mainly because using outlines was an already adopted and perfected style, but they did exist.

The GBC port of 1942 is one game with no outlines, at least for sprites. With the improvements in technology, outlineless sprites became less favoured. Outlines did look nicer on a smaller screen but with a wider palette of colors available, they remained an "NES thing" . ## Modern Homebrew and DMG In modern Game Boy development, we can test styles and different device looks through emulators. Let's begin with one modern (upcoming) game that features outlines, Fields of Eutomia. I'm using shaders to emulate the DMG and CGB screens on RetroArch. Not quite the same thing, but it gets pretty close to the real deal! This is not a particularly busy scene, but you can see how the black outline helps distinguish the skin of the player from the lightish background. It stands out, although this depends on the style of the environments. A style different to the background can help with readability, but be careful not to make it too different, that'll hurt consistency. The next game I'll show is Zirconia, a platformer by SpicyFuse. You can play it here. This game decided to go with the outlineless approach while also being compatible with DMG. As you can see, black is used for the gloves, bottom & outline of the helmet, and is colored purple on GBC. Since the backgrounds of the game are mostly black, these parts are not usually visible on DMG. This is a good example on how to design sprites if you decide to go outlineless, *make the less important parts the ones you'll see less.* Now that you have some ideas about graphics, what will you choose? Outlineless might be more risky, but it also looks more unique. The early NES style might be just as charming as that which came after it. But that's just one opinion, what are your thoughts? SōdoDev

SōdoDev

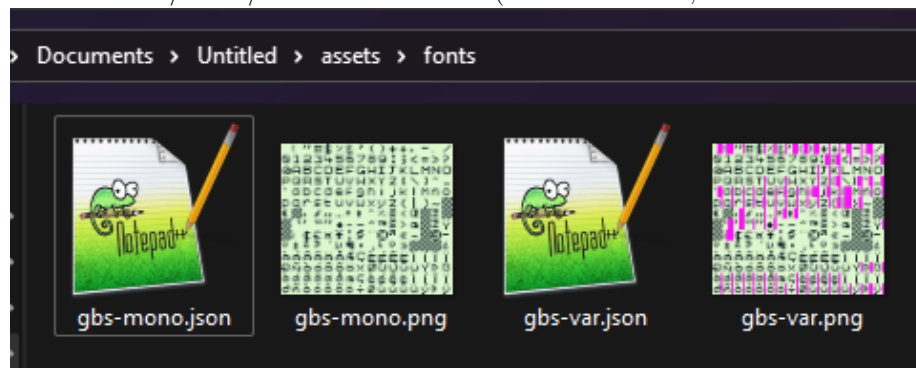
a/k/a "Anima" – Game Developer Wannabe, Average Internet Enjoyer, Human Being. (he/him) ### Share this:

- [Click to share on Bluesky \(Opens in new window\)](#) Bluesky
- [Click to share on X \(Opens in new window\)](#) X
- [Click to share on Facebook \(Opens in new window\)](#) Facebook

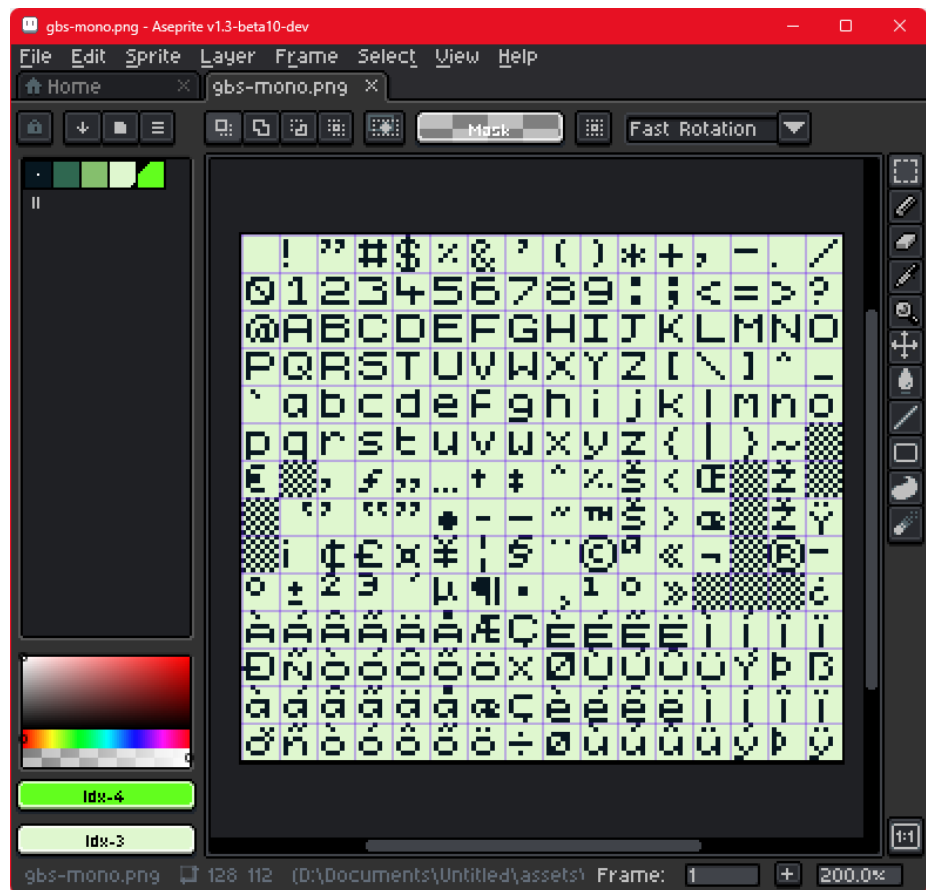
- Click to share on Mastodon (Opens in new window) Mastodon
 - Click to share on LinkedIn (Opens in new window) LinkedIn
 - Click to share on Reddit (Opens in new window) Reddit ### *Related*
-

Understanding Fonts

Having text in games is usually pretty important. It can be used for a HUD, a UI or an entire story. GB Studio includes tools to display typed text, and it comes with a font (and a codepage) to render it. This font is located in the `/assets/fonts` folder since 3.0 (for 2b5 or earlier, it's in the `ui` folder).



There are actually two font files, but it's the same font with a variable-width variation. In this article, we will take a look at how fonts work and what you can do with them, plus some tricks! ## How do fonts work in GB Studio? Font files in GB Studio are saved as an image file with the `.png` format. Each letter is divided into `8x8px` tiles, and they're laid out in a `16x14` grid (`128x112` pixels). The arrangement of the letters follows the extended ASCII standard, ranging from the `32-255` (`20-FF`) values for it. The first chunk is for normal characters used mainly in English, and the extended chunk is for special glyphs and symbols, mainly used in other European languages. These can be changed to suit your needs. Prior to 3.0, there was only one font available, so users who wanted to use customized characters usually modified the bottom half for their purposes. Thankfully, a GB Studio project can now hold up to 255 different fonts. This is very useful, as you can use one font for all kinds of text, and another one for storing special characters and symbols. ## How do I modify a font? Modifying a font is super easy, just open it with an image editor, one that can use grids preferably. I use Aseprite, but anything can be used.



Fonts support all 4 colors, as they're part of the UI. They cannot however use transparency. But, there is a fifth color that's pretty useful. Using the color #FF00FF (pure magenta) will allow you to create variable-width fonts. Paint to the right of a character the area that will be removed when rendered on screen (it's a good idea to leave at least one pixel width for separation/clarity). The font



gbs-var shows this.

It may break the perfect tile spacing, but it helps fit more text into less space! Fonts can have any name, but make sure it's a .png! ## What about codepages? Codepages are a nice little feature, especially for languages that don't use the Latin alphabet. They allow you to map a letter to a character in the font. They're saved in the same folder, and have the same name as their fonts. The default fonts include templates for codepages, in .json format. Open one of them in a text or code editor like Notepad++, and let's see what we can do with them.

```

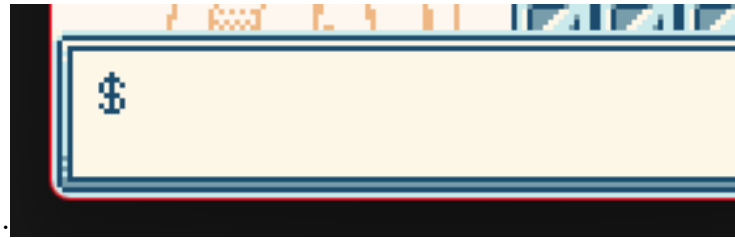
1  {
2    "name": "GBS Mono",
3    "mapping": {}
4  }
5

```

JSON file | length: 42 | lines: 5 | Ln: 5 | Col: 1 | Pos: 43 | Unix (LF) | UTF-8 | INS

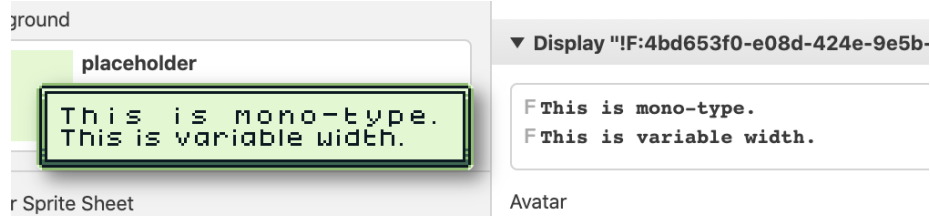
...there's not a lot here. We can start making a codepage by counting to the character we want to use from 32 onwards. For example, let's map this weird Kanji that doesn't exist in real life (𐤎) to the dollar sign. To do that, add a line like this, just remember to put it inside the mapping brackets.

```
{
  "name": "GBS Mono",
  "mapping": {
    "𐤀 ": 36
  }
}
```

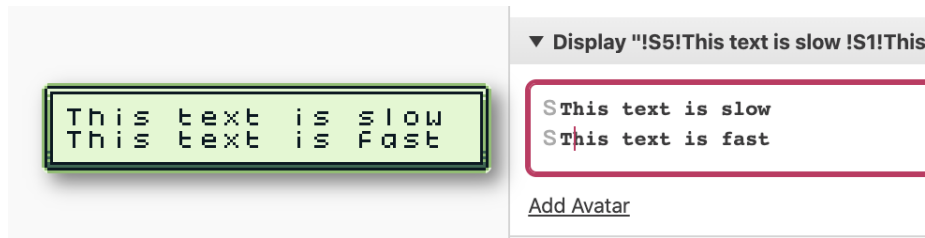


If we now load this font and type that character...

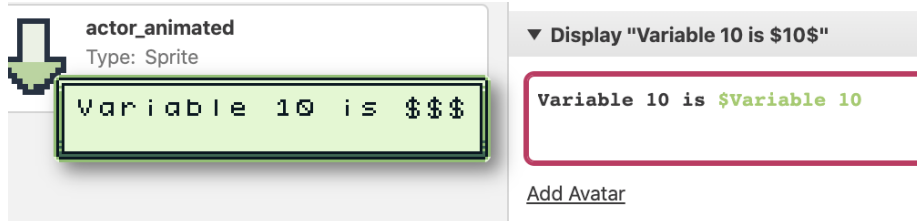
We get a dollar sign! If you want a better example, several people have made codepages for their purposes. For example, here's a Japanese font with a codepage by Amaitorte! Supports Hiragana, Katakana and uppercase Latin! ## Wildcards and Escape Sequences Wildcards are another neat GBS feature, and Escape Sequences are actually from GBVM, but they can also be used with regular dialogue events. Let's take a look at Wildcards first. There are three wildcards: \$ for variables, # for characters and ! for speed/fonts. To call one of these wildcards, simply type their sign followed by a letter or number for a menu to show up. This menu lets you select a variable for \$/# and a speed or font for !. Wildcards will show up differently depending on which one you use. You can edit them by clicking on them to bring up a drop down menu, and remove them the same way as any text by using backspace/delete. Using an exclamation point (!), start typing the name of the font to have it appear in the drop down. A capital "F" will appear in the event window to denote the font change.



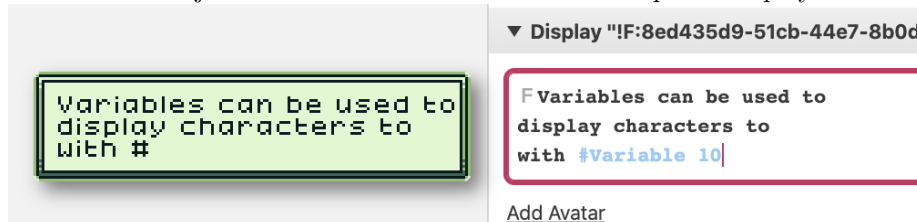
To change the speed, type "S" after the exclamation to see the different speed values. They are Instant, and Speeds 1-5, the higher the number, the faster the text will render. A capital S will appear to denote the speed change in the event window. Keep in mind that Instantly rendered text actually takes more CPU to run, so some lag might be noticeable, especially on DMG systems.



The dollar sign wildcard (\$) will print the value of the variable. Start typing the name of the variable after the dollar sign to bring up variables with those letters in the name in the form of a drop down. You will see the variable name highlighted in the event, but in the text preview it will just show “\$\$\$” – this will be rendered correctly when the game is run.



If I set the above variable to the value of 5 for example, the resulting text box would say “Variable 10 is 5” . This is really useful for debugging, but also good to display information to the player. Finally, the hash symbol (#) works differently. Instead of the variable rendering, it will display the corresponding character from the ASCII chart. For example, if I wanted to display the letter “a” , I would need to have the corresponding variable set to a value of 97 (remembering the mapping from earlier). “abc” therefore would need three separate variables of values 97, 98 and 99. This can be used in conjunction with other tools for text input like player names.



Escape Sequences are part of GBVM, but most of them can be used within dialogue events for the same effect. Here are the different escape sequences you can try inserting into a display dialogue event and their different effects. | | | |
 *** ## V. Advanced Techniques & Scripting - Advanced Trigger Techniques - Basics: Custom Events - Basics: Math Expressions - Basics: Math Functions - Basics: Variables - Creating Cheat Codes - Dynamic Auto Scroller - GB Studio Music Lesson 6: Advanced Drums - How to Use QR codes to Enhance Your GB(C) Games - How to: Multi-Language Mechanic - Incrementing and Decrementing Values Like a Pro - How to Use The Min/Max Functions to

Save Time - Learning GBVM - Making an Isometric Game in GB Studio -
Modifying the GB Studio Engine - Screen Sized Bosses - Slopes in GB Studio -
Tile Swapping 2.0 - Translate your games with GlyphBoy Swap - Understanding
GBVM - Unraveling The Thread Events - Upgrading the GBS Platformer
Engine - Variables and Maxing Them Out

Advanced Trigger Techniques

A GB Studio scene with multiple triggers setup. ## Trigger Happy Triggers in GB Studio are essential tools. They allow players to access the next scene, collect objects, enter zoned areas for gameplay, and more. Within these orange boxes lie innate abilities that transcend their original intentions. There are some unique ways of tapping into their power that have remained unseen until now.

These new concepts and uses will seem trivial at first, but when you finally dive into the usefulness of them in your projects, you' ll wonder how you ever developed without them! ## Platformer Position Checks To effectively check where a player enters or exits an interactive zone, we can use triggers to determine their exact height or horizontal position in the level, providing us a means to gatekeep certain gameplay elements. For this example, I will be referring to the skateboarding sections of Grimace' s Birthday, because without triggers, these sections would have been nigh impossible to create in GB Studio. Showing the skating trigger script in GB Studio.

A skating trigger within Grimace' s Birthday' s first level I used triggers for skateboarding to determine if the player was above the railing when they entered the trigger, and if so, allowed them to grind on that surface. The trick to this technique is to use a **[Store Player Position In Variables]** event in the On Enter section of the trigger and switch the units from Tiles to Pixels. Then assign the X and Y values stored by the event to specific variables. I prefer to label them something specific like "x_position" or "y_height."

When you enter the trigger, it will now give you a numerical amount stored in the Y position' s height variable, and then you can use that to check with an **[If Variable Compare With Value]** event to determine whether the player is coming from above or below. This technique is greatly improved by either using the built-in Debugger to check the values of the watched variable or a non-modal **[Display Dialogue]** event that displays your variable with the \$ symbol before (example: \$y_height).

Keep in mind that if you use a math comparison with your variable value, a lower number results in a Y-axis position higher on the screen, so check for less than or equal to your value to check above, and greater than for below. You can also use this to check left or right positions as well, which can be useful to determine if a player is entering or exiting a trigger into a new zone from the left side or right side. Make sure you do multiple tests when debugging these triggers to get an approximation of where the player is and use that data to arrive at the value you check for. ## Users and Interfaces When creating a

menu system, you have a couple of options. One option uses the player as a cursor and allows them to select different actors to activate or deactivate them, which can be tricky to ensure they can interact exactly where the elements are, but might be a stylistic choice that works better for your game. Another option is when you're allowing the player to move from one element to another using the D-pad, and it's typically best to use triggers to accomplish this.

As the player is thrown from trigger to trigger, using [**Attach Script To Button**] events to define what each direction does gives a more streamlined experience and provides a faster way to traverse interfaces. You can also use these Button events to attach what each option does. For example, using the A button to select that option, while the B button will use a [**Set Actor Position**] event to place them right on the exit trigger, giving them one last chance to pick whether they want to exit the menu or not. An advanced script setup for a menu system using triggers.

A menu system using the A button to select and the B button to move the player to the Back option ## Cutsscenes and Other Similar Uses Since the entry and exit of triggers run their scripts before giving control back to the player, you can also use this exclusivity to ensure events happen on the entry or exit of your trigger before you move to another trigger or area. This can be quite useful for cutsscenes and other long-form events to be able to pause and run code in a sequence, similar to the On Init script when loading in a scene.

A quick note about using this “freeze” is the animation of the player typically continues to run their current animation state. A separate idle or single-frame animation state used exclusively during cutsscenes might be a good way to alleviate this awkward visual glitch. Switch back to your previous animation state when you are done with your sequence. ## A Trigger in Disguise Triggers do not react to actors or projectiles. Actors in the latest versions of GB Studio also don't react to collisions with each other. That's a tough reality, but something that makes defining areas for your actors to move and fire off projectiles within even more challenging to script around. Additionally, while it's not a common problem, you might also find yourself in trouble if you run out of triggers in your scene or if you need a trigger that updates your location as you move through it. In those situations, you might want to use an actor with a [**Launch Projectile**] event instead of a trigger to serve as a fake one.

Using an actor this way requires some patience, but it is possible to use them as a means to define actions happening within a certain zone, especially if you need one that doesn't confine to 8-pixel squares or if you're checking for other actors or projectiles. You will need to edit the collision boundary box inside the sprite for the projectile you are launching, create a blank sprite to set this up, making it the size you need for your situation, and ensure your [**Launch Projectile**] event's speed is set to “None.” Keep in mind that you need a unique sprite for each shape you want to set up this way, as you won't be able to change the Collision Bounding Box of the projectile at run-time. Setting up a fake trigger actor.

Using a Launch Projectile event's sprite to set up our fake "trigger" actor. An example would be using it to see when an actor moves into an area to execute its events, say if it runs into a trap that would hurt them. It will then execute every time there is a collision between the actor and the projectile by running the events in the *On Hit* section — make sure both the actor and the projectile are set up to interact with each other first in their Collision Group settings!

Once you set this up, it's worth noting that it will check for the collision every frame, so you may get some undesired results or slowdown. Enter the *logic gate* solution.

While it's not an end-all solution — mostly because using an actor as a trigger doesn't allow you to check when it is no longer touching another actor or player without using an **[Attach Timer]** event — using a logic gate will make sure that it either executes that action once, or at the very least, not immediately over and over. An example of a logic gate.

An example of hit code that uses this "logic gate" concept. You can use this technique by checking with an **[If Variable Is 'False']** event, which should be its default state, and then immediately setting it to true using a **[Variable Set To 'True']** event as you run your code. Once it is finished, you can either use an **[Attach Timer]**, a **[Wait]** event or just let the events naturally end to set the variable back to false again.

These logic gates are not confined to use within triggers, or using actors as triggers, but are useful any time you want to ensure that code will only execute when you want it to. Feel free to use them wherever it makes sense!

With this final trick in our arsenal, it's easy to see how many different ways you can use triggers or actors acting as triggers to solve difficult problems. Hopefully, these tricks and techniques will help you utilize some unique ways with issues where an area or position needs to be checked, or many other situations. Triggers are great tools, so be sure to maximize their potential! Pearacidic

Pearacidic

Writer, Game Designer, Engine Bender, Retro Enthusiast. Co-creator of the In The Dark series and Grimace's Birthday. (He/Him)

pearacidicgames.com/ ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### Related

Basics: Custom Events

As you continue working on your game, you may notice that several scenes have a lot of scripts in common. Perhaps you run events setting a player's sprite sheet and coordinates at the start of a new area, attach the same actions to your buttons, or a series of events to give out rewards after a battle. When you run into these situations, you probably find yourself copying and pasting the same events over and over again. Sure, placing them in an event group makes the cloning between actors/triggers/scenes more manageable – but when you need to make a bug fix or optimize a sequence, you'll find yourself losing time tracking down each copied and pasted script.

Luckily, there's an event type to make repeating scripts much easier: Custom Events! # What Are Custom Events? Custom Events are groups of events that you can create and add to your actor, trigger, and scene scripts just like normal events. The trick is that unlike the other standard events in GB Studio, *you* define what a custom event does. Once you've created a custom event, you can re-use that script multiple times, and whenever you make a change to a custom event, it will update for each occurrence across your game. Custom Events make managing repeated sequences much easier, as you only need to look in one place to test, edit, or fix a bug.

If you're creating the scripts anyway, how is this any different from simply creating the scripts directly in the actor/trigger/scene? While a Custom Event's biggest advantage is its ease of editing, there are some other uses. Let's dive into our first example to show what can be done. # Building a Custom Event For our first example, we'll build a custom event to replace a feature that has been “removed” from later versions of GB Studio. With the release of the 2.0 Beta, Actors no longer have a Movement Type, which removed the frequently used “random walk” option. Since that change, creating a randomly walking NPC or Enemy now takes a Math Event, A Switch, and several Move Relatives in an Actor's “On Update” tab. Sure, it's not the longest script, but it's also unlikely to change for each NPC you're using it in. This makes it a perfect candidate for a custom event. Instead of Copying and Pasting those events into another actor (or even cloning the actor altogether), we can create a “Move Randomly” custom event that will be called from the search list whenever we need to make an NPC move randomly.

To start, navigate to your Custom Events section. In Beta 5, you can find Custom Events via the Navigator (on the left side). If you've hidden this, you can go to **View > Show Navigator** to bring it back. If you're on earlier versions, Custom Events appear when you click the background of the Game World (similar to how you view a player actor's settings). Custom Events can be given a name, and also a description that can be used as a reminder of what the event does.

Once created, you'll notice that the custom event field looks just like any other script. The major difference is when you add your events, variables and actors

do not reference any that have been placed in your game. Instead, custom events use placeholder values, labeled alphabetically from A to J.

A single Custom Event can have up to 10 variable and 10 actor placeholders. When an event is added to a custom event's script, you can select which placeholder to reference. These placeholders can be renamed, and once that Custom Event is added to a scene, event or actor, then the **actual** variable or actor you wish to use can be assigned to that placeholder.

To continue our example, let's create that Walk Randomly custom event. Create a new Custom Event, and name it "Walk Randomly". In its script, add a [Variable: Math Functions] event. Select Variable A, and then have it set to a random number between 1-4. Then, add a switch, and set it to check variable A. In each of its "When" sections, add a move relative in a different direction with collisions enabled for "Actor A". An image of the Custom Event being created in the GB Studio interface. An image of a script written for a custom event. It is a Switch event with multiple options. In your scene, create a few actors, and then in their "On Update" field, add an event and search for "Walk Randomly" – you should see your custom event listed. When you add a custom event, it has a list of each placeholder variable. For this event, you should see a Variable A placeholder and an Actor A placeholder. Assign the variable to Local 0, and the actor to "Self" (these should be the default anyway). Do this for each actor in your scene, and then test out your game. An image showing how Custom Events can be searched in GB Studio interface.

Custom Events show up by their names. An animated scene in GB Studio. 3 characters are moving randomly, using the custom event.

That's too fast! (unless they're dancing) Oh no! They're moving much too fast. We'll need to make a change. Go back to your custom event's script, and place a "Wait" event at the end of it (a few seconds should do). Now, when you test your game, the change should be reflected for all actors that were using that custom event. A screenshot from GB Studio, showing a custom event being edited to now include a Wait event.

Add a wait to the script... An animated scene from GB Studio. The 3 actors are now moving slower.

That's better. # Limitations Custom Events are an excellent way to optimize your game when you find yourself repeating scripts. The quality of life they add to bug fixing and editing scripts across scenes is unmatched, but they do have their limitations. Due to their limit of referencing 10 variables and actors, larger scripts (for example, being used in rpg battle systems) may need to be split across multiple custom events. You also need to make sure you're properly re-assigning the placeholder variables and actors once you add your custom event to an actor/scene/trigger. While they do improve the QA process for scripting, they have their own areas that will need to be reviewed for mistakes when adding them to scripts. # Other Common Uses This article covered how to use a custom event to replace a feature that is commonly used across game types, but there is

no way to know exactly what scripts are being repeated in your game and how converting to custom events could help you in your development. In my own experience, I've had success with custom events being used to handle:

Button Assignment: attaching scripts to customize button mappings (like weapons, jumps, start menus, etc) can be handled via a custom event that is run at the start of every scene.

Managing Money: when working with variables over 255, running the scripts to pass values back and forth between your ones/tens/hundreds columns can be quickly referenced in a custom event, reducing your script complexity

Changing a player sprite sheet: if you have a character customization or use 4-frame animations, custom events can be called at the start of the scene or on update to make sure you're always using the correct sprite sheet.

Controlling Random Encounters, Puzzle Rocks, or other common elements of a dungeon in an RPG.

Updating HUD variables and displaying actors in an RPG battle system. Dan Enders

Dan Enders

Friendly Neighborhood Game Designer, Podcaster, and (ex)Librarian (he/him)
GB Studio Games | Tabletop Games ### Share this:

- [Click to share on Bluesky](#) (Opens in new window) Bluesky
- [Click to share on X](#) (Opens in new window) X
- [Click to share on Facebook](#) (Opens in new window) Facebook
- [Click to share on Mastodon](#) (Opens in new window) Mastodon
- [Click to share on LinkedIn](#) (Opens in new window) LinkedIn
- [Click to share on Reddit](#) (Opens in new window) Reddit ### *Related*

Basics: Math Expressions

We're entering the summer season, and as the sun comes out and the weather warms up, there's only one thing on everyone's mind: math! School may be wrapping up, but that doesn't mean you're done with classes yet.

There have been many changes to features and functions in GB Studio since our last entry in the Basics series, and today we'll be covering some of the new functions introduced in GB Studio 3.0+. ## What's changed for Math Functions? Earlier in the Basics series, we covered the many ways you can manipulate your variables with standard events like incrementing or decrementing variables, as well as the more robust [Math Functions] event. If you're new to GB Studio, you may want to review that article first. If you're already familiar with these events, you're in luck, very little has changed for the core math functions in GB Studio 3.0+. The main change that has impacted math

in newer versions has been the addition of 16 bit variables, which increases the minimum/maximum values of your variables. Where necessary, older articles in the Basics series will be updated to make note of which functions are no longer limited to the previous cap of 255. ## New for 3.0: Math Expression Events GB Studio 3.0 introduced the following new Math events: * [Evaluate Math Expression] * [If Math Expression] These two events can be powerful new tools to add to your scripting. By using Math Expressions, you can write a single formula to handle math that previously may have required multiple stacked events to run! ## Writing Math Expressions Both of these new events allow you to write a “math expression” to manipulate your variables. An expression is a combination of numbers or variables and at least one mathematical operation. For example, “2 + X” (or, in GBStudio terms, 2 + \$Variable 1), is an expression.

Operations are the functions we use to manipulate numbers such as addition, subtraction, multiplication, etc. In GB Studio, we can use the following operations:
| | | | ***

Basics: Math Functions

As discussed earlier in our Basics series, Variables are a core component of making a game in GB Studio. To use variables to their full potential, it’s important to know which events you can use to manipulate variables, and the limits to each event’s usage. There are 15 events in GB Studio that directly deal with a variable’s values, not including more advanced events that store elements in variables. Four of these events are connected to the concept of Variable Flags, and will be covered in a future article. In this article, we will review the standard variable events, comparing variables, and the multiple features found in the Variable: Math Functions event. # Standard Variable Events ## Variable: Increment by 1 / Variable: Decrement by 1 Incrementing a variable increases its value by 1 each time the event runs, to a maximum of 255. Decrementing a variable decreases its value by 1 each time the event runs, to a minimum of 0. Incrementing and Decrementing variables can be a quick way to manage variables that track items or simple counters, and the built in clamp can be quite convenient. However, Incrementing and Decrementing are limited by only being able to adjust a variable’s values by 1 at a time. ## Variable: Set to True / Variable: Set to False / Variable: Set to Value Variable values can also be set directly. An important consideration for working with values is that True and False are not special settings for a Variable, they are simply binary values. When a variable is 0, it is false. When a Variable has a non-zero value, it is true. Setting a Variable to True changes its value to 1, while setting it to False changes its value to 0. Setting a Variable to a Value changes a variable from its current value to the new specific value. Setting variables to true and false are useful for tracking simple states (whether something has been interacted with, if an objective is complete, etc). Setting variable values directly is useful for managing variable states at the beginning of a game or scene, such as setting an HP value. # Math Functions for Variables All of the standard variable events

can be replicated in a single event: [**Variable: Math Functions**]. This event requires more input to work, but also introduces more options to all the events listed previously. Using the first drop down field on the left lets you choose the function for the event. The second drop down on the right provides the options for this function (more on these options below). The 6 functions for a Math event are:

- * **Set To:** Sets or defines your variable
- * **Add:** Adds to the variable
- * **Subtract:** Subtracts from the variable
- * **Multiply:** Multiplies the variable
- * **Divide:** Divides the variable
- * **Modulus:** Runs a Modulus function on the variable (more on this below!)

With Variable Math, you can Add, Subtract, Multiply, and Divide the values of a Variable. These functions operate exactly as they do in normal mathematics, but there are a few GB Studio concepts to take note of when using them:

- * Unlike with Incrementing and Decrementing variables, when using Variable Math, a Variable's values will **wrap** if they exceed 255 or go below 0. If you want to limit your variable's values to the minimums and maximum values (0 and 255), you'll want to check the "Clamp Value between 0 and 255" check box in the event.
- * Division in GB Studio will always round down. One of the benefits of using math functions over standard variable events, is that you can specify the values you'd like to add/subtract/etc. This gives you more options than incrementing or decrementing by 1, but you are not just limited to working with Values. Using the second drop down box on the right, you can choose from 5 options when using a math function.
- * **True:** This option is the same as a value of 1.
- * **False:** This option is the same as a value of 0.
- * **Variable:** This allows you to select another variable for the math operation to be performed with.
- * **Value:** Allows you to enter a specific value to use.
- * **Random:** Allows you to enter a range of values, and GB Studio will use a random value from within that range. Note: The ranges are Inclusive. For example, using a Math Function event like the one below is the same as using an "Increment Variable by 1" event, because we're only adding 1 to the variable and we're clamping the result at 255. The value can be set to any number however, allowing for far more flexibility. You could instead use the "Random" option to add a random number: Note: When using the random options, be sure to review our article on how GB Studio handles randomness, as the behavior may appear different than what you expect. And when you start using multiple variables, you can combine your variables to do new things. Let's say you have a battle system and you want to increase your player's damage based on their strength stat. If you're tracking the player's stats separately, you can modify the damage in your battle system using math events: While the "Set To" option changes a variable's values directly just like the "Set Variable to Value" event, using it in a math function gives you more control over what you set a variable to. Just like with addition/subtraction, etc, you can set a variable to True/False, a set Value, a random number, or even another variable's value! The final option you have for Variable Math is the **Modulus** function. A *modulo operation* divides one number by another, but it returns the **remainder** of the division. For example, performing a normal division operation of 95 by 10, the result is 9.5 (in GB Studio, this would display as 9 since the engine always rounds down). If you were to use a Modulus operation:

95 mod 10, it would display a result of 5 since there is a remainder of 5 left over from the division. For another example, 12 mod 10 would result in 2, and so would 12 mod 5. This may sound complicated at first, but it is a very useful function when using variables to store information. For example, how many coins a player has: 99 coins can be thought of as 9 tens and 9 ones, and a modulus function can check the ones value. Don't worry if you don't see an immediate application for Modulus in your game just yet, more advanced tutorials will make use of this function. # Variable Checks Once you've started working with and manipulating variables, you'll likely want to use them to check if certain conditions have been met. By using the events to check your variables, you can control which scripts run in your actors, triggers, etc. The most straightforward ways to check variables are to see if they are True or False. This is done using the [If Variable is True] or [If Variable is False] event. Remember, all variables are false if their value is 0 (and all variables start at 0), and a variable is true as long as its value is a non-zero number. The second way to check your variables is to compare them. There are two compare events: Variable Compare to Value, and Variable Compare to Variable. Both events have a similar setup, the event compares the current value of the selected variable, and checks if it is: * **Equal to** a Value/Variable * **Less than** a Value/Variable * **Less than or Equal to** a Value/Variable * **Greater than** a Value/Variable * **Greater than or Equal to** a Value/Variable Comparing variables has many uses! You can check if a player has found a certain number of items, if an NPC has been talked to a certain number of times, if a player can afford to purchase an item from a shop, etc. Comparison events are the backbone of many RPG battle systems, too. If you find yourself checking the same variable with multiple comparison events, you may want to use a switch in its place. You can learn more about switches in *this article*. If you want to download the ROM and GB Studio Project file we created for the demonstration in this article, check out our Itch Page. Dan Enders

Dan Enders

Friendly Neighborhood Game Designer, Podcaster, and (ex)Librarian (he/him)
GB Studio Games | Tabletop Games ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### Related

Dynamic Auto Scroller

This tutorial was developed using GB Studio V3.1.0 and is applicable to that specific version. So you want to make an auto scroller but avoiding spikes and

pitfalls is too bland. You want a massive boss that chases you. But how do you do it? What smoke and mirrors does the Game Boy or GB Studio have? In this two part tutorial I will discuss how to implement these key concepts:

- * Moving the camera independently of the player
- * Detecting player position relative to the camera to create dynamic 1D collision boundary
- * Utilizing the window/overlay layer and invisible sprites to create a screen-sized moving boss

Part 1 – Auto Scroller ### How-to auto scroll the screen This is fairly simple in concept, you have to decouple the camera from the player and move the screen using the inbuilt events. Follow these steps to set that up. Add a scene and in it’s “On Init” tab, add a [**Lock Camera to Player**]event. Uncheck the axis you’ll be scrolling. This will result in the camera no longer following the player on that axis. Next, add a [**Move Camera To**] event, and set it to where you want the camera to initially be located. Now add an actor, pin it to the screen and hide it. This will act as the “updater” of the camera position On that actor’s On Update tab, add a [**Move Camera To**] event, and set it to the desired end position. When you run the game, the camera position is initialized, then scrolls to the target position using this script. The player will be able to freely leave the screen, so it’s up to you to place collision boundaries.

Dynamic Collision Boundaries As of writing there are no built-in features that allow for changing collision during gameplay, and unfortunately this is not a particularly easy task to get working without issue. GB Studio is designed to use pre-defined collision detection rather than dynamic edges based on game conditions. To achieve dynamic edge collision, you can compare the player’s position to the edge of the area you want them to collide with and then and adjust the player’s position (push back) with movement to ensure the player remains within bounds. The [**Store Actor Actor Position in Variables**] event retrieves the player’s position in pixels but there is no equivalent for camera position. Instead GBVM can access two variables, “_camera_x” and “_camera_y”, which correspond to their names. Let’s put that into practice and create two lines of GBVM code which store the position data into variables named **CX** and **CY** (for Camera X and Camera Y)

```
; get the camera position and store them in global variable
VM_GET_INT16 VAR_CX, _camera_x
VM_GET_INT16 VAR_CY, _camera_y
```

Calculating the screen boundary The variables _camera_x/y are relative to the center of the screen only when the camera axis is decoupled from the player, otherwise it will be the player’s location. Here’s four different math expressions you can use to calculate the edge of the screen according to the camera’s position. | | | | | ***

How to Use QR codes to Enhance Your GB(C) Games

We all know we can’t create the sickest looking cutscenes for our Game Boy games. Sure, you can try to do some wizardry and get as close as possible to

a fluent little clip, but at the end of the day it will be at best a rather fancy PowerPoint presentation, and that's OK!

But what if there was a way you could overcome the limitations of the Game Boy?

In this article I'll explain how you can use QR codes to enhance your game's experience. An example of a QR code displayed in GB Studio. ## QR codes for what? You can link a video, audio file or website to your game to enhance the experience for your players.

You could add some details to your story with the help of a cutscene uploaded to YouTube. A link to an audio file could be played as a background atmosphere of a crime game. Or hell, you could even link to a website where your players need to solve a riddle, get a specific code that they then enter in your GB game to play the next chapter of your mystery game.

For our GB Studio GB Pixel Jam 2023, we used a QR code to allow you to go to the submissions page of the Jam, so you can read the pixel artists' comments about their work.

There are countless possibilities of using QR codes in your games. But you need to always be aware that there can be situations where your game could potentially become unplayable, for example if someone plays your game without access to the internet, or without access to a device that can scan QR codes.

So in general, QR codes should be used to enhance the experience of your game, not to be a crucial part of it. Nothing could be more frustrating for your players to play a game that they are then locked out of, because they can't access the QR code for whatever reason. ## How to implement QR codes First we have to create the QR code. There are many free-to-use online QR code generators out there. I used QRCode Monkey for this article.

You type in the URL you want to link to, click "create QR code" and download it as a PNG image file. On this specific generator, you can use the slider underneath the QR code to select the lowest quality (200x200px). An image of a QR code being generated from the qrcodemonkey interface. We now have a nice looking and fully functional QR code like this as a PNG file. A QR code. Scanning this will take you to a music video. Next step would be to create a background for your game in which you add the QR code.

I'm using Aseprite, but it will of course work in any other graphic program.

Step 1: Open the QR code in your graphic program, select everything, and copy it. A QR code being edited in Aseprite. The QR code links to a music video. Step 2: Paste the copied QR code into your GB Studio background. An example of pasting a QR code from Aseprite into GB Studio. Step 3: Resize the QR code to fit the GB(C) screen (160x144px). An image of a QR code being resized in Aseprite. Step 4: Open your GB Studio Project and add a new scene An image showing the GB Studio interface. A QR code will most likely be over the

tiles limit of a standard scene. So we have to use the “logo” scene in this case. An image showing GB Studio warning the user when a scene has gone over the tile count limit.

GB Studio will warn you and provide a tile count when your scene exceeds tile limits! Step 5: Select “Logo” as your scene type An image showing how to set a GB Studio scene to a Logo scene type. If however you want to have a bit of UI or other things going on in your scene while showing a QR code, you can simply resize the QR code to be even smaller, since they are pretty solid to use. You should always test if the code still works with your own smartphone though, just to be sure. I recommend not making it smaller than one fourth of the GB screen. An image showing a smaller scene being made in Aseprite. In this case, you could also use another scene type, since the QR code will likely fit inside the tiles limitations. An image showing how the new scene now conforms to GB Studio’ s tile limitations for a scene. A bigger QR code is always the better way to go, since it’ s easier to scan even under non-ideal light conditions (thinking of original GB(C) devices without backlight). A photo of the QR code being displayed on an actual Gameboy.

A photo of the smaller QR code being displayed on an actual game boy. ## Extra tip: Using your own URL and Subdomains URLs of third party platforms can always change. If YouTube someday in the future decides to rename themselves and get rid of their old URL, then you would have a game with dead-ended QR codes in them that nobody could use anymore.

Or you just have to upload another version of a video which has a different URL, or you want to link to a different place on the internet for some reason.

In these cases, using your own URL could strengthen the longevity of your introduced links.

In the example for a cutscene on YouTube, you would create a subdomain, for example cutscene.yourgameboygame.com, which you would set to redirect to the YouTube URL of your cutscene. Now you can create the QR code of your subdomain cutscene.yourgameboygame.com and still be able to change the place it’ s redirecting to. Martin Cremerius-Gauer

Martin Cremerius-Gauer

Martin first went into making GB Studio games with his playable portfolio of his own personal website. He is generally interested in a very broad band of topics, and adds new things to the skill list all the time. Being born in the late 80s, the original Game Boy DMG-01 was his first gaming console, and so he kind of has a nostalgic romance with everything around it.

2ndlwgames.com ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook

- Click to share on Mastodon (Opens in new window) Mastodon
 - Click to share on LinkedIn (Opens in new window) LinkedIn
 - Click to share on Reddit (Opens in new window) Reddit ### *Related*
-

How to: Multi-Language Mechanic

In modern games you typically select your language at the beginning, or it is detected automatically and the system selects the language.

The limitations of the Game Boy are present in this topic as well, as developers had to figure out other ways of supporting multiple languages. A common practice is to release the game on different cartridges with one language per cartridge / rom. So you had “Pokémon Blue” for English-speaking folks, and “Pokémon Blaue Edition” for German-speaking folks (the one I had as a kid).

Since space on cartridges is very limited, there would’ ve been no way of providing more than one language for Pokémon games on one cartridge.

But if your game is smaller in scale (like a short puzzle or arcade game), and you can afford the space on the cartridge, then it’ s a different story.

In this article, I’ ll guide you step-by-step through the process of creating your own multi-language mechanic to support multiple languages on one rom.
Step 1: Preparation First you should decide which languages you want to support.

In my example, and in my game “Fuse Dash,” I decided to support English, German, French, Portuguese, and Spanish, to cover a large portion of people around the world.

If your game is for Game Boy Color only, then you can even work with country flags (which are harder to differentiate with the 4 colors DMG palette).

Create a language select background in the pixel art program of your choice. A GB Studio scene background created in Aseprite, providing 5 language options.

Creating a scene using Aseprite. Now create an actor sprite that you can use as your cursor.

For this example project I used the arrow that comes with every new GB Studio project (as of version 3.2.1). GB Studio’ s sample arrow sprite. It’ s best to select “Fixed Direction” in the animation type of the sprite, so you don’ t have to handle actor facing directions later. The GB Studio sprite editor. Selecting the ‘Fixed Direction’ animation type for an actor in GB Studio. Now you can create a new scene, select your language-select background and your cursor actor.

The scene type can be “Point and Click,” but it should also work as Adventure or Top Down 2D type. Setting a scene type to “Point & Click” in GB Studio.

Step 2: Cursor movement When we run the game, the cursor actor can move freely.

So we need some mechanics to restrict the movement to be inside of our list of selected languages.

First, we add the UP movement mechanic.

Add “Attach script to button” to the scene and select the UP movement. Inside you’ ll have to add an “Actor move relative” script and add -2 to the “Y” value. Searching for an event in GB Studio. Entering the coordinates in a Move Relative event in GB Studio. Now wrap the previously added “Actor move relative” inside the Else of an “If Actor at position” script, which you set to the highest place your cursor should be able to move up.

In the case of this example, that’ s X:2 and Y:5.

You can use the red indicator while increasing or decreasing the values.

Why use the Else in this script? Because we only want the cursor to move 2 tiles up, if it’ s not at the highest position already. A screenshot of the input script being written in GB Studio. An additional screenshot of the input script being written in GB Studio. Now do the same for the Down input. You only have to edit the “Actor move relative” Y to 2 (instead of -2), and edit the Y value to represent the lowest point the cursor should be able to move to. In this case it’ s Y:13. Creating more input scripts for our menu scene in GB Studio. And as the last step for the cursor movement, we need to restrict the cursor in moving left and right.

Add another “Attach script to button” script to the scene and select Left and Right. Also make sure to tick that “Override default button action” checkbox. Selecting multiple inputs with the Attach Script to Button event. Now your cursor should only be able to move from line to line in a vertical manner, without going above or below the list. The language selection screen we created in GB Studio, with the option for English displayed. The language selection screen we created in GB Studio, with the option for French displayed. The language selection screen we created in GB Studio, with the option for Spanish displayed. ## Step 3: Language selection mechanic We now need a variable to store which language was selected.

Create one and name it accordingly. In this case I named it “language.” A screenshot of the variable details in GB studio. Add yet another “Attach script to button” script and select A as your selection button. You can of course also use another button, like B or Start, or straight up all of them if you like. The Attach Script to Button event in GB Studio. Now we’ re again using the fixed positions of the cursor to determine which language is selected.

We’ ll start with the first language, which in this case is “English” and would be selected if the cursor is at X:2 and Y:5. Using the Player at Position event to run a script for our cursor at different positions. Add a comment field and

write down which values represent which languages. Creating comments in a GB Studio script. Add a “Variable set to value” script, set the variable to your language variable (again, in this case I’ ve called it “language”) and set the respective value. Setting the value of our variable with an event. Now do the same for all other languages. Copy the “If player at position {2,5}” script, place it inside the previous ELSE state and change the X and Y, as well as the language variable values accordingly. Using an IF statement to control which variables are set at which cursor positions. Expanded image showing the rest of the script we have written. Now place a change scene event at the end of the “Attach script to button” for the language select button so it changes the scene when the player has selected a language. Adding a scene change event in our GB Studio script. ## Step 4: Showing the right dialog Now add a new scene, or go to the scene in your game where you want to show a dialog. Creating a new scene in GB Studio. Add a “Switch” script, change the “Number of options” to 5, and let the numbers start with 0 instead of 1. A screenshot of a Switch event in GB Studio with 5 options available. In this example, 0 is used for English, 1 for German, and so on.

So if the language variable has the value 0, it should show an English dialog. If it’ s 1, it should show German, and so on. A screenshot of a Switch event in GB Studio. The first option has English dialogue, while the next has German. And that’ s it, you now have a language select within your game! Selecting English from the language selection screen we created for our game. A GB Studio scene with English text. Selecting German language from the options screen we created. A GB Studio with German text!

Martin Cremerius-Gauer

Figure 1: Martin Cremerius-Gauer

Martin Cremerius-Gauer

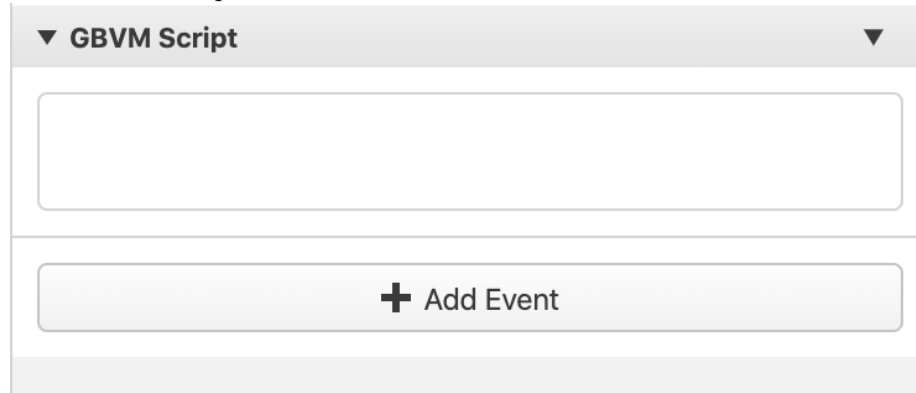
Martin first went into making GB Studio games with his playable portfolio of his own personal website. He is generally interested in a very broad band of topics, and adds new things to the skill list all the time. Being born in the late 80s, the original Game Boy DMG-01 was his first gaming console, and so he kind of has a nostalgic romance with everything around it.

2ndlawgames.com ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### *Related*

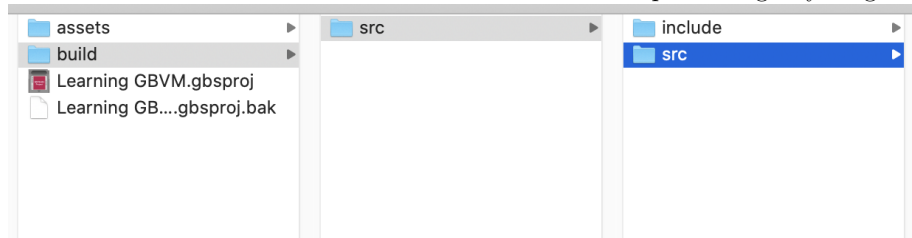
Learning GBVM

GBVM (Game Boy Virtual Machine) is the core of GB Studio 3’ s powerful tools. While it’ s entirely possible to make an excellent game in GB Studio without GBVM, knowing how to use it unlocks a lot of potential (like Animating Background Tiles) for more advanced game development. We did an in depth explainer into how GBVM works in the backend, and now it’ s time to detail how a GB Studio user can learn to code with GBVM Script. # The GBVM Script Event While it’ s possible to use GBVM in other ways (more on that later), most users will take advantage of it using a **[GBVM Script]** event. There are no parameters or options, just a big empty box for you to type in the commands. There is no limit to how many characters you can use in one script, so there’ s no requirement to split up commands into separate events. However, depending on how you intend to use and debug your scripts, it may be worthwhile to split up the different parts of the code for organization’ s sake. It’ s also possible to go all in with GBVM and code almost everything with it, but some math events and other commands are easier and more practical to execute with the “stock” events in GB Studio.



The GBVM Script event # The GBVM Syntax Mimicking Assembly language, GBVM is fairly relaxed with syntax (at least compared to other programming languages I’ ve used). Here are some of the basics to keep in mind when using GBVM for the first time. Comments are denoted with a semicolon (;) and any characters that appear after one in a line are ignored by the compiler. Use comments to break up and mark down your code and leave yourself notes to make it easier to read and debug later. Code is executed line by line, and each line of code is ended with a line break. When using GB Studio, the text wrapping can make it appear to be multiple lines, but only a line break will end the current instruction. Accidentally hitting return/enter on your keyboard mid command and adding a line break will cause an error when it comes to compiling. All GBVM commands start with the prefix VM_ followed by the command name. Commands typically follow a “Action, Where, What” logic, but certain commands also allow for extra parameters. There’ s a few other neat aspects of GBVM, but that’ s all you need to know

for now. So where does one start when wanting to learn GBVM? Here's a few different ways you can do that. # **Tutorials** This may seem obvious, but there are quite a few tutorials already out there thanks to the efforts of the community. We released the Animating Background Tiles article as a teaser for what GBVM can do, and there are some other great ones on Itch, GitHub and on GB Studio's Discord that can be found. If you have a question about GBVM, there's a specific channel on the Discord where you can ask and get help, and we've seen some impressive stuff come out of discussions there. # **Source Code** All of GBVM's instruction set can be found at <https://github.com/chrismltby/gbvm/blob/master/include/vm.i> – this is what's used to actually build the GBVM code when GB Studio is compiled into an app. There's not a lot of explanation on how the commands work, but brief descriptions do give a good start. A more detailed list of commands can also be found at <https://gist.github.com/pau-tomas/92b0ad77506088d184a654af226f5b7d> thanks to Pautomas. That list is a work in progress, but the explanation of how many commands work is an excellent start. # **Exporting Game Data** This is my preferred way to learn GBVM. In the menu bar, select **Game > Advanced > Export Project Data**. This will take all the information, scripts and code that's in your game and export it into the GBVM intermediate code that's actually used to build your ROM. If you already have a complex game with multiple scenes and scripts, this will most likely look like a bunch of gibberish and will be hard to parse. I recommend starting with a fresh empty project with only one scene and one event. For this example, I made a single scene and added a [Variable Set To 'True'] event, and selected the Global "Variable 1" for simplicity. After exporting the project data, a folder appears in the Project's "build" folder called **src** (stands for "source"). There are two main folders, "include" and "src" and within both are a series of files and folders pertaining to your game.



The "include" folder contains .h files. These are "header" files which contain definitions for the code, and can largely be ignored when learning GBVM. Inside "src" is where you'll find the interesting stuff. There's a lot of code to be found here and you can learn a lot about how the Game Boy, GBDK and GB Studio treats commands and variables, but for now we'll turn our attention to the file named "script_s0_init.s".

```
_script_s0_init::          VM_LOCK
    ; Variable Set To True          VM_SET_CONST          VAR_VARIABLE_1, 1
    ; Wait 1 Frame          VM_PUSH_CONST          1          VM_INVOKE
```

b_wa

```

; Fade In          VM_SET_CONST_INT8          _fade_frames_per_step, 1          VM_FADE_IN
; Stop Script      VM_STOP

```

If it wasn't apparent, this is the first scene's script that runs On Init. There's a bit going on here, but thankfully some helpful comments have been generated to make it easier to parse, and since our Variable Set To True event is the first one that gets executed, it shows up high in the list. This shows one of the simplest commands to use in GBVM. Let's break it down:

```
;Variable Set To True
```

As stated before, this is a comment, and any text that follows the semicolon gets ignored by GB Studio when your game is compiled. This text is for you, the developer to know what's happening in the code.

```
VM_SET_CONST
```

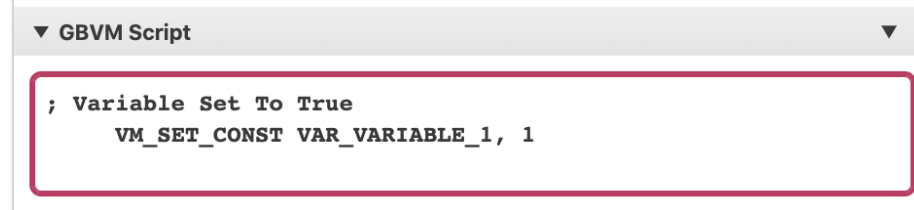
Since this starts with VM_ we know it's the actual GBVM Command. It 'sets' a given variable to a Constant (const) value, that is to say an actual number value and not a pointer to another placeholder or variable.

```
VAR_VARIABLE_1
```

Something new! Hopefully this is apparent, but it's the Variable 1 we selected in the GB Studio event. Global variables are called this way, using the prefix VAR_ and the variable name in all caps, substituting spaces with underscores.

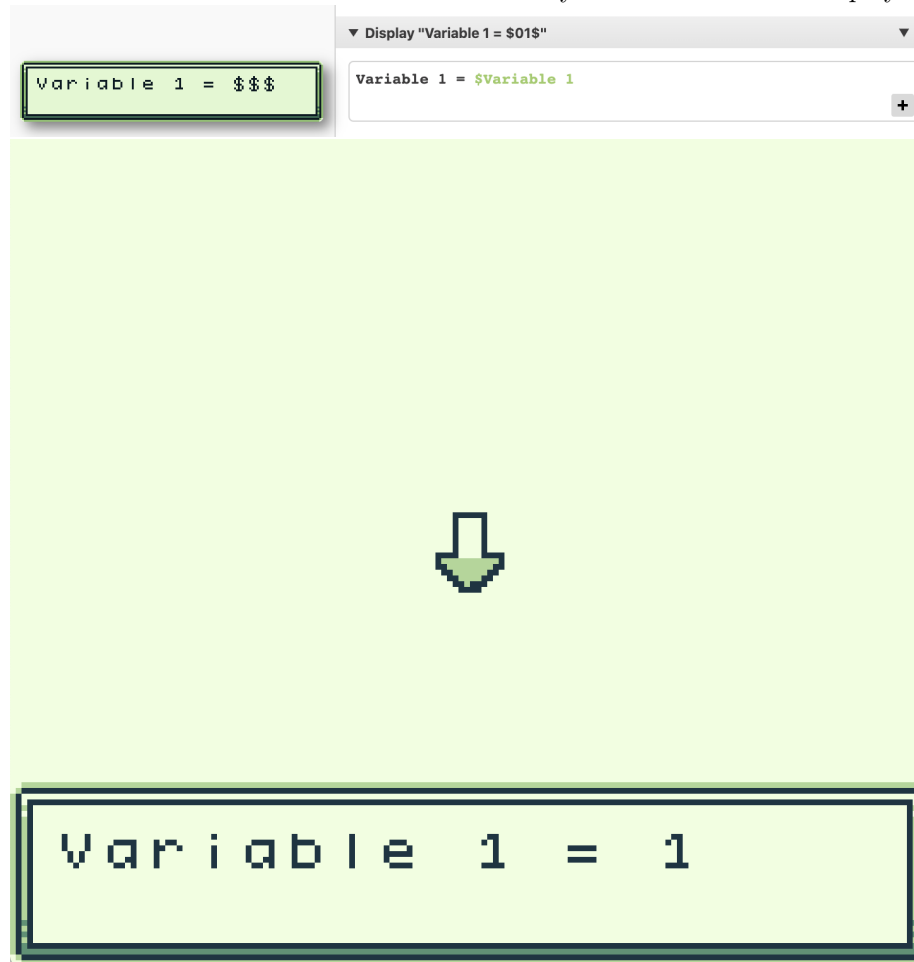
```
1
```

The value that the variable is set to. In GB Studio, all variables start with the value 0 which gets interpreted as False when using an "If" event, and any positive number is interpreted as True. When setting a variable to True, it simply sets it to 1. Neat, eh? So what can we do with this information? Let's try using this command into a GBVM Script event in GB Studio and see how it behaves. Add a [GBVM Script] event at the end of your On Init script. You can then either type out everything yourself, or simply copy and paste those lines into the event.



A space is needed between the command and parameters, but otherwise whitespace is ignored so feel free to add or remove spaces to your liking to make it easier to read. Remember that text after the semicolon is ignored, so the Command needs to be on a new line. I've left the comment in there to make it easier for myself. This command doesn't do much by itself, so add a [Display Dialogue] event to confirm that it's functioning. Typing in a dollar sign (\$) followed by a variable name the text window will bring up a drop

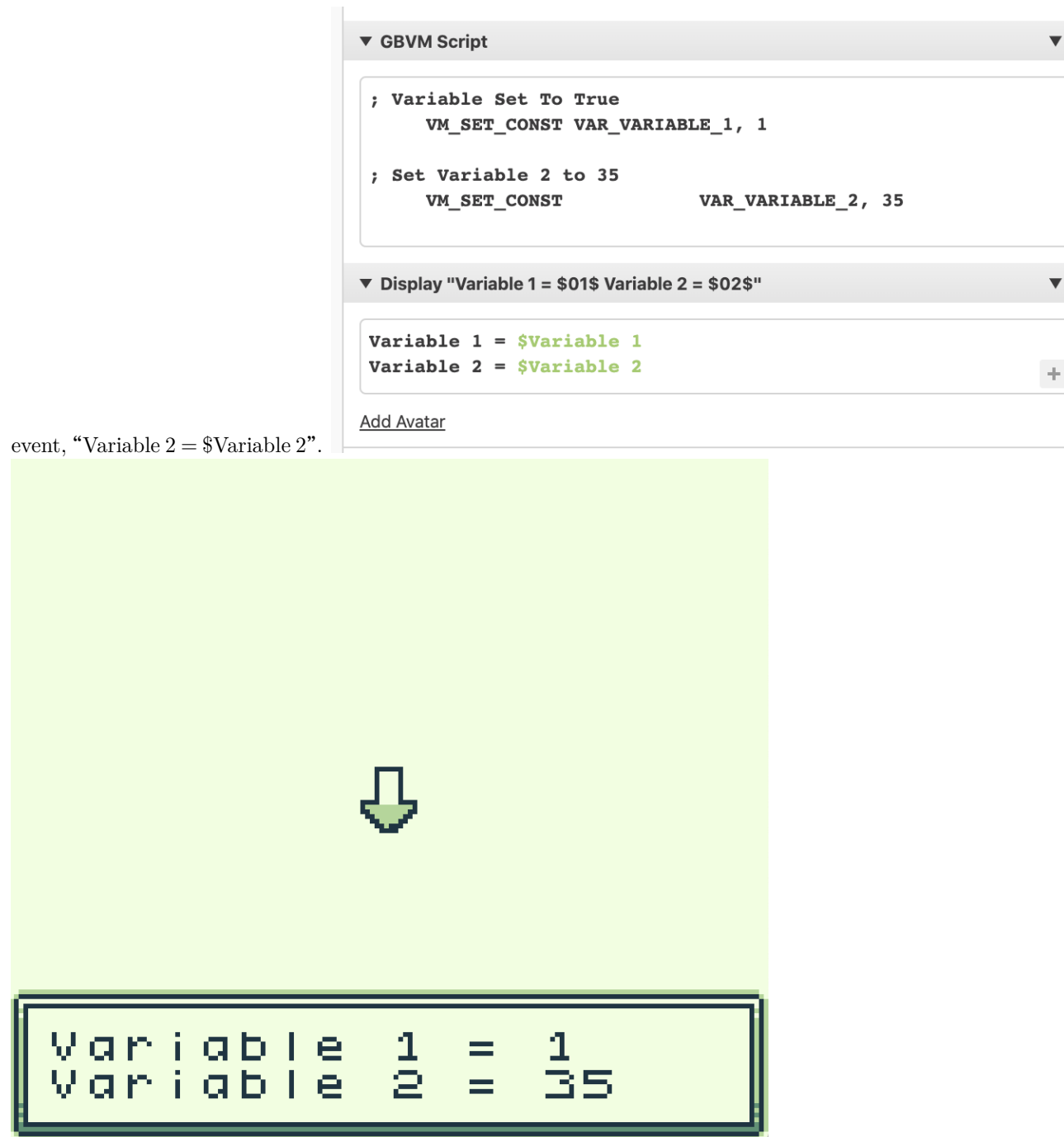
down, select “Variable 1” to display its value. I like to write it out fully as “Variable 1 = \$Variable 1” so that it is clearly labeled when it’s displayed.



It works! Sweet! Our GBVM script is working with no errors. If you do see one appear in the build window when it’s compiling, check your spelling and spacing/line breaks. Let’s see if we can replicate the code with a different variable and value for a different result. I’ll keep adding lines of code to the current GBVM script event instead of adding a new one. This time, let’s change Variable 2 to the value of 35, just so we can see some double digits.

```
; Set Variable 2 to 35  
VM_SET_CONST          VAR_VARIABLE_2, 35
```

To confirm that it’s working, I’ll add another line of text to the Display Dialogue



Feels great when the result is expected. Woot! It worked as expected. Notice that we've been doing everything within GB Studio events? The Exported

Game Data does not work backwards, so you cannot edit code there and expect those changes to be reflected within your game. It may be possible in a future version, but for now the Export Data is a way for you to see what’s going under the hood of your game when you add an event and inspect other aspects of the code. You can use Export Project Source and compile it yourself into a ROM, but that’s beyond the scope of this tutorial and moves beyond working with GB Studio – although it’s also possible to combine those exported data files with other scripts and tools outside of GB Studio, that will have to wait for another day. Let’s try exporting the Game Data again and see what’s changed since we’ve added these events.

```

; Variable Set To True
VM_SET_CONST          VAR_VARIABLE_1, 1
; Variable Set To True
    VM_SET_CONST VAR_VARIABLE_1, 1

; Set Variable 2 to 35
    VM_SET_CONST          VAR_VARIABLE_2, 35

; Text Dialogue
VM_LOAD_TEXT          2
.dw VAR_VARIABLE_1, VAR_VARIABLE_2
.asciz "Variable 1 = %d\nVariable 2 = %d"
VM_OVERLAY_CLEAR      0, 0, 20, 4, .UI_COLOR_WHITE, ^/(.UI_AUTO_SCROLL | .UI_DRAW
VM_OVERLAY_MOVE_TO    0, 14, .OVERLAY_IN_SPEED
VM_DISPLAY_TEXT
VM_OVERLAY_WAIT        .UI_MODAL, ^/(.UI_WAIT_WINDOW | .UI_WAIT_TEXT | .UI_WAIT_BT
VM_OVERLAY_MOVE_TO    0, 18, .OVERLAY_OUT_SPEED
VM_OVERLAY_WAIT        .UI_MODAL, ^/(.UI_WAIT_WINDOW | .UI_WAIT_TEXT)/
; Stop Script
VM_STOP

```

Conveniently, GBVM Script events get put into the Game Data verbatim, another reason to comment your code well. Also notice that the original “Variable Set To True” Event is still present at the top of the script, which has now become redundant. But wow, that’s a lot of lines of code for what seems like a simple dialogue window. It is possible to take that code, copy and paste it into a GBVM Script and edit some of the lines to display what you want, but in this scenario is it worth it for the result? I would argue not, and using the GBVM event makes more sense. You’ll often find other events like Math Expressions that require many more lines of code in GBVM as opposed to a single event in GB Studio. So why use GBVM? # Why Use GBVM? It turns out there are a number of features within GB Studio and GBVM that don’t have an event or interface to call on. We already showed how Background Tiles can be swapped using GBVM, and the reason it’s not available as a “stock” event is due to the challenge in representing all that data in an easy to use event interface. The development team behind GB Studio has opted so far to provide a

more approachable experience for beginners than overcomplicating things. Those who wish to explore more advanced tools have the ability to do so with GBVM Scripts. However, even in a simple scenario like above, you could see the benefit in using GBVM for setting a complex or large number of Variables. Instead of many Variable events, you could have one GBVM Script with many lines of (hopefully well commented) code. There's a tonne more to learn about GBVM, including other commands that are exclusive to it and how they can be used, but we'd rather create actionable tutorials than abstract concepts and theory. Expect more tutorials from us in the future that take advantage of GBVM. In the meantime, try out GBVM for yourself and maybe play with exporting Data to see how events translate and what you can do with it. Happy coding! Emi Paternostro

Emi Paternostro

Audio Engineer, Mac Technologist and Video Game Developer. Managing Editor of GBStudio Central. (he/him)

proximitysound.com ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### *Related*

Making an Isometric Game in GB Studio

Isometric games have been a popular genre since the 8-bit era. They provide a unique perspective that allows players to view the game's world spatially, while still utilizing simpler 2D graphics on less capable hardware. Games like *Atelier Marie*, *Qbert**, *Kirby's Dream Course*, and *Final Fantasy Tactics* show how this perspective lets players interact and move through the game world in a more dynamic way. However, isometric movement can be challenging to implement in GB Studio. GB Studio 3.0 added the ability to use pixels instead of tiles for movement events. In this tutorial, we will create our graphics, and use this addition to set up our controls. # **Setting up the scene** Let's create a background that our player can walk on. In your art program of choice, create a 160×144 image and draw a diamond shape whose lines follow a **2 pixel, 1 pixel pattern**, as pictured. Scale plays a role here, so for this example we will build to accommodate a 16×16px sprite.

This single tile can be used to build out a complete tileset. Feel free to use this one! Use your single tile to create a floor by duplicating it. Be mindful of the grid and keep everything aligned properly. I've added the cardinal directions here for the sake of the tutorial. # **Setting up the animation states** Next, let's

set up our player. Use a three direction sprite sheet similar to the player sprite sheet in the GB Studio example project. We'll set up five Animation States for this example: an idle and our four directions. This is necessary because we need to manually call them when scripting the movement later. In GB Studio, open the Sprite Editor. Select the sprite on the left sidebar you intend to use for the player. In the bottom left pane, you should see the default sprite state that GB Studio has created. In the right sidebar, Change the Animation Type to **Four Directions**. Make sure **Flip Right To Create Left Facing Frames** is checked unless you want a unique left facing idle. Fill out the canvases for each of the animations. This will be our default idle state. Next, let's set up states for our individual cardinal directions. I'm using cardinal directions here, since isometric operates in pseudo-3D space and it is a more illustrative description than "right-up" or "down". In the Animations pane in the bottom left, click the '+' symbol to add a new Animation State. In the right sidebar, you can rename an Animation State and change the Animation Type, which changes the type of animation you can assign. Rename the state to "North" by hovering over the left side of the Name field and clicking the pencil icon. Change the Animation Type to **Fixed Direction**. Add your walking frames using the up-facing sprites. Repeat the steps to create the rest of the renamed Fixed Direction states for each of the other directions. Each cardinal direction will correspond to a direction on the D-pad. Create your frames for each state as follows: **North** for up facing, **east** as right, **west** facing left, and **down** for south. Alternatively, you can copy and paste to create a duplicate state from the one selected. Click the state you want to copy from the Animations pane. Click the triangle in the top right of the sidebar to bring down the context menu. Click **Copy**. Bring down the context menu again and click **Paste**. Rename your duplicate state and change the sprites accordingly. For facing left, you can use the tiles for facing right and flip them using the flip horizontal button. Select the sprites on the canvas you want to flip and these options should appear in the right sidebar. # **Setting Up Controls** Now that we've done all the preliminary work, let's bring it together by attaching the states to the controls and set up movement. In the On Init of the scene, add an [**Attach Script To Button**] event set to **Left**. Check **Override default button action** to override the default controls. In order to have the player move while holding the directional button, input needs to be checked on every frame. Attach Scripts only work on the first initial press and do not continue to check for input. Adding a [**Loop**] inside of the script fixes this. Create a new Loop event inside the Attach Script to Button event. Inside the loop, place an [**If Joypad Held**] event and set it to **Left**. Now, when left is pressed and held, the game will continue to check this script. If Joypad Held is made up of a True section and a False section. The True section is for everything we want to happen when the player holds the button assigned. The False or **Else** section is used for what we want to happen when the player quickly taps or releases the button. In the True section, add an [**Actor Move Relative**] event inside the If Joypad Held event. Set this to **Player**. Set **X to -2** and **Y to 1**. Setting X to a negative will move the actor to the left while setting Y to a positive number will move an actor downward.

Combined, this will move down and to the right diagonally. |||| **

Slopes in GB Studio

If I had to pick the most common request in the GBS discord, it would have to be none other than the release of GBA Studio. But if I had to pick a runner-up to that request, it would be the ability to create platformer slopes in GBS! In early November of 2023, I put out a bounty on the code required to allow GBS users to finally do just that. The goal was to find a way to make platformer slopes not only function on the DMG without impacting performance negatively, but also allow users to paint slope collision tiles in the scene editor with relative ease, and then release this plugin to the community at-large for all to enjoy. Within only a few days, there was such a strong response to this call to action, that it was clear a dedicated discord server would need to be created to manage the influx of contributors, contributions and the discussions surrounding them. “The GBS Slopes Initiative” was born!

Thanks to the generosity of coding wizards such as Canight and Gearfo, whose contributions have formed the foundation of a new and improved slopes integrated platformer.c file, to the support of members of the GBS dev team such as Rulz and Pautomas, and also to Hauntology (who’s Platformer+ plugin needs no introduction) Within just a couple of weeks, slopes in GBS went from dream to reality. An animated screenshot showing early versions of slope collisions being used in GB Studio.

An early test of slope collision running in the sample project file on DMG Our progress soon caught the attention of Chris Matlby, creator of GBS. He too joined in the community collaboration, expressing an enthusiastic desire to not only offer his time and knowledge, but to merge this code into the official branch of GBS3.2! After some time spent on quality assurance and further implementation of a user-friendly dedicated slopes paint tool, we are proud to present the end result: slopes – by the community, for the community! ## **How To Implement Slopes in GBS** Applying a slope collision tile is as simple as painting-in any of the various collision tiles, and will automatically snap to either a 22.5° or 45° angle. Simply select the *slope brush* tool from the *collisions toolbar*, click-and-hold on a tile of your choice, and drag the mouse in the direction of your slope. Releasing the mouse button will lock your slope collision in and you’re done! An animated screenshot showing how slope collisions are drawn in GB Studio.

Couldn’t be easier! Some additional features to keep in mind include: * Hold *Shift* while creating a 22.5° slope to offset it by half a tile (4 pixels). * Hold *Ctrl* to quickly switch to drawing vertical or horizontal one-way collision using the slope brush tool (holding *Ctrl* and *Shift* together will flip the one-way collision tiles direction). * *Right Click* using your mouse to quickly delete existing collision tiles. ## **The Slopes Initiative Keeps Rolling On** What you see in GBS3.2 is the tip of the iceberg as far as slope integration is concerned. Down the

line, the GBS dev team will assess how best to expand on slopes based on the contributions of charitable coders such as Gearfo and others in the community.

Hauntology is also working on an all new version of Platformer+ that will allow for slopes compatibility with his own plugin. Platformer+ is fully compatible with GBS3.2 in its current state, however adding P+ at this time *will remove slopes functionality* as it overwrites the platformer.c file. **## A Special Thanks** Within only a few short months, platformer slopes have become a reality for everyone to enjoy. To see so many talented people put their minds together and achieve such a goal so quickly has been an inspiration.

Thank you to Gearfo and Canight for all their hard work; to the GBS dev team including Pautomas, Rulz, and Chris Matlby; to all of you who joined my discord to help solve coding problems and provide feedback and suggestions; and to those flies on the wall there to watch it all unfold in quiet support.

The Slopes Initiative shows us that the GBS community can accomplish great things when we work together. It makes me wonder what other lofty challenges can be overcome and shared freely to the benefit of our community. Why not head to the GBS discord and suggest some ideas for further initiatives you'd like to see explored in future? With a little time and effort, no slope is too steep! Gumpy Function

Gumpy Function

Independent Games Designer, Artist, Film Enthusiast and Full-time Dad (he/him). Check out my games here!

www.gumpyfunction.com/ **###** Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit **### Related**

Unraveling The Thread Events

Common Thread

Thread events are one of the newest additions to GB Studio, and with no documentation to explain how to use them effectively, there has been a lot of confusion as to what they can and can't do. At their core, they are the functions running behind the scenes, starting and stopping events and executing other scripts. With the release of version 4.1, we now have direct event-based access to them, so let's dive into the ways they operate and how they can make your game's scripting even better.

To properly understand what the Thread Start event can do for you, we should look at some of their close relatives first. Timers and the Script Lock and Script Unlock events are very similar in nature to how Threads work. Each of these events can do some amazing stuff in context, but they certainly have their limitations, which you may or may not be familiar with. Since Attach Timer Script is the event closest to what we can accomplish with Thread Start, we' ll look at that one first.

Attach Timer Script and its options Timers are a great way to run code periodically, based on a set timer amount. They execute the entire script every x amount of seconds or frames, depending on what you choose to define them. However, one of the biggest flaws of Attach Timer Script events is the inability to remove the Timer once it has already begun executing the script you assigned. It remains inaccessible to cancel those functions until they are complete, somewhere out in the digital aether. Because of that fact, using Timers in some projects is quite frustrating, so the best practice is to make sure you limit the length of what is being executed on any of your Attach Timer Script events. Since Threads are tied to a variable and executed based on that variable' s value, they do not have this limitation.

Next, we have Script Lock and Script Unlock, which are two of the most useful events in our repertoire, especially when you are looking to set up cutscenes and ensure the player or other actors are unable to move in the meantime. These pause and unpaue all scripts, restricting player movement, On Updates and more, until the execution of the script you locked is unlocked again, or the script finishes.

Locking ensures that you have exclusive access to what is happening with your code and it executes as you desire, but Script Lock and Unlock have to be meticulously planned to ensure that exclusive access. Unlike Threads, these events by themselves are not their own scripts, requiring you to place them within another script, whether it' s a custom script, in an actor' s On Update, or a Thread Start event itself.

Now that we know a bit of how Threads relate to these other events and what they can do, we' ll look at the Thread Start and Thread Stop events in detail and get into the nitty gritty of what they can and cannot be used for. ## Weave Nothing Behind Threads are scripts that run similar to the built-in On Init or On Update scripts, but they can be called at any time. They also execute from top to bottom, which is the same as any other script filled with events, and can even run simultaneously, if started one right after the other from the same source. While there are a lot of things Threads can do, we should touch on what they are limited by to help set expectations.

First off, Threads don' t naturally run continuously. I could think of a lot of situations where that would be nice if that was built-in, but many more where I am thankful that they don' t. They are typically intended to be a one-off script execution, but that' s not to say that they can' t be continuous, it just takes a

bit of effort.

To run a Thread in the background over and over, you will need to utilize a Loop event inside of it. Loop for and Loop While events are the most commonly used, but if you're utilizing a lot of variable values for determining how your Loop event runs, you can use a Loop While Math Expression event too.

An example of a Loop While event used inside of a Thread Start event While it's looping, the Thread event will continue running in the background until you either give it the condition needed to stop, or use a Stop Thread event. Keep in mind that as long as you are running a Thread, it's the same as having another constant script from an actor running in the background, so stop it when it is safe to do so, if not only to help with performance but also to help you stay organized. Thankfully, threads don't continue to run after finishing running through their events or when switching scenes, so doing so will automatically stop all running threads.

Because Threads utilize a unique variable, it is best practice to set a couple of them aside for exclusive use with them. Naming them based on what they do might be an easier way to keep track, for example: "Soldier 1 | Thread" and "Soldier 2 | Thread." If you try to use one variable for multiple Thread events, you will quickly find out how they fight for priority and this will mess up your execution order, including stopping events unintentionally. It's best to play it safe and either set them up with Local Variables from where the Thread event was called, or set up Global Variables to use. ## Two Threads Are Better Than One One of the best features of Start Thread events is the ability to run them simultaneously. Instead of relying on the order of sequence when setting an event to true and praying that the actor On Updates pick up the command at the same time, you can simply start threads one after another and sync them up that way. The best use for this convenience is having multiple actors moving at the same time.

Often when you have cutscenes running, you want to have multiple things moving at the same time. With Threads, you can now achieve this with a level of control that we did not have before. A couple of examples would be having two NPCs walking side-by-side, or complex animation sequences for visual effects. There are a lot of uses for this feature, but it's up to you to figure out what works best for your game.

It's worth mentioning that even though Threads are a dream come true for a lot of developers utilizing cutscenes for a cinematic approach to their games, Threads cannot run while a modal event, such as a dialogue box, is running. To get both to work, you would need to ensure that the dialogue box is set to non-modal, which means it will not close when it is done rendering the text inside or react to a button press.

Setting up a Non-Modal dialogue box in the Display Dialogue event You can then close this box with either the Close Non-Modal Dialogue or Hide Overlay events, which you can add to any button with the Attach Script To Button event. If you

wanted to add the functionality back to using a button press to close the dialogue box after it finished typing out in the box, you could use either an Attach Timer Script to set a wait time or Thread event with a Wait Event to add the ability to close it with a delay. This also gives you the long-awaited ability to set up a button that allows the player to skip cutscenes that run in the background, and then you can remove that functionality on the next Attach Script To Button event. It's tricky, but most of the cleverest means of doing something are. ## Tying It All Up Once you have settled on a means of handling your Timer scripts and Buttons, it would be wise to encapsulate that code into a custom script if you plan on using it throughout your project. You cannot access any actor other than the Player if you place a Thread inside of a custom script, however, so plan accordingly. This is something I call double-nesting, as the engine does not provide access to actors this way. You will notice you have the same limitations for Attach Script events as well, but these can be placed in an On Init script and still have access to other actors.

My last bit of advice regarding Thread events is to use them as “dropped code.” To explain this better, let's look at triggers. Triggers execute events in order and freeze everything going on in the scene until they are complete. Let's say you want the player to be able to use the trigger as a button to move a block in the background, but you need to show some visual effects to really sell the sequence. If you added anything with a time-based element, like a Camera Shake event, or a literal Wait event, the scene and more importantly the player actor will be frozen in time, leaving them unable to move.

Instead, you could use a Thread Start event to “drop” this code within the events of the trigger and leave your player able to continue moving through the zone. These are also quite useful when executing code from On Hit events, especially long sequences of code. It's situations like these that really allows the use of Thread events to shine.

Threads are a great new tool that allow us to do some amazing things. They are familiar to other scripting tools that we already have access to, but allow even more freedom to script things with precision and care. Why not give them a try by implementing in either your current project or your next one and see how much easier they make the scripting process. Pearacidic

Pearacidic

Writer, Game Designer, Engine Bender, Retro Enthusiast. Co-creator of the In The Dark series and Grimace's Birthday. (He/Him)

pearacidicgames.com/ ### Share this:

- [Click to share on Bluesky \(Opens in new window\) Bluesky](#)
- [Click to share on X \(Opens in new window\) X](#)
- [Click to share on Facebook \(Opens in new window\) Facebook](#)
- [Click to share on Mastodon \(Opens in new window\) Mastodon](#)
- [Click to share on LinkedIn \(Opens in new window\) LinkedIn](#)

- [Click to share on Reddit \(Opens in new window\)](#) [Reddit ### Related](#)
-

Variables and Maxing Them Out

Choosing to make a game on the Game Boy is about limitations. You're faced with them at every turn and your game gets shaped by them.

Want an extra frame in your run cycle mid way through development? Better find some details you can remove from the background tiles (potentially across every level). Choices between a second enemy type on a level or an extra collectable, with sprite, tile, and palette limits to stay under. Juggling this with the constant concern of slowdown from the ancient CPU.

When I began to plan out my current project, Fall from Space, I was aware that I could be pushing against a number of hard limits. File size is something I've been constantly monitoring, checking the size against my estimate of percentage left, but also the overall project's variable count.

GB Studio allows 500 global variables to be set, so from the start I was very conscious to minimize the use of these, using flags where I could but also local variables where possible.

I thought I was doing fairly well, but I made one major mistake.

Local variables compile as global variables.

This I didn't plan for, and if your total variable limit, global and local combined, is over 768, you can run into problems; variables can start to overwrite other variables and you may have issues with saving and loading the game. A screenshot from Fall from Space. This is something I became aware of in the QA testing of Tales of Monsterland DX, but thankfully it wasn't far from this limit, so I could easily bring it under. However, I made a (slightly concerned) note to check how Fall from Space was shaping up and it was at – **1056**/768 variables. ***screams***

1056 and the game had a lot more content to come.

So, what can you do if you're in this situation, or even better, to avoid this situation? ## Finding Your Projects Total Variable Count in GB Studio From the GB Studio menu, choose Game > Advanced > Export Project Data.

Then from your project folder, open include/data/game_globals.h A screenshot showing where the MAX_GLOBAL_VARS variable is defined in the project source code. Search for MAX_GLOBAL_VARS. This is the number of total global and local variables, combined in your project.

If you need to buy yourself a few more variables, it can apparently be bumped up to 800 without problems, although I'd recommend trying the steps later in this article to get this down as low as you can.

For this, choose Game > Advanced > Export source. A screenshot showing the GB Studio source code and where you can edit the MAX_GLOBAL_VARS variable. Open build/src/include/data/game_globals.i, search for MAX_GLOBAL_VARS, if the number is higher than 768, it can apparently be bumped up to 800. ## Reducing Your Global Variable Count After the sinking feeling in my stomach had subsided a little, I found two areas of the game that I thought were the main offenders, where I knew I had definitely used a ton of local variables: ### Enemy AI Perhaps unwisely for a system with limited resources, enemies, rather than travel on a set path, monitor and react to the player's position. If the player is to the left of an enemy, it'll turn around and head towards them. When the player is close to the enemy, it'll telegraph and then attack.

This was all done mostly, with a set of multiple local variables and a custom script.

To improve this, 7 sets of reusable global variables were created for the enemy AI; 7 is the maximum number of enemies currently in a scene. It was tedious to replace all these across the game, but 1056 variables became 780.

Still over the limit with a lot more enemies planned, but better. ### Coins Fall from Space is an open world, non-linear game, you can visit and revisit levels to explore and find secrets. Coins operate in a similar way to XP, being used to purchase upgrades and items.

As levels can be revisited, coins needed unique variables to mark as collected or not, and there's no way around this (if your game is linear, you can make these into reusable values and save even more).

Previously, coins neatly used their own local variable for "collected" and a custom script triggered when the coin was collected (that fed in the coin value to allow for a variety of sizes). They could then be copied and pasted quickly without modification, remaining unique. However, having just a few coins on each scene can very quickly add up to a lot of variables, and it's simply unsustainable for a larger game.

I didn't want to drop coins or vastly reduce them, so the best compromise I could find was to create new global variables using flags, e.g. Coins_1A (planet/set group). Each coin within that group has its own ID that I could reference in a new custom script to mark a coin collected.

Flags allow 16 on/off switches within a single global variable, so I figured it should be a sizeable (16x) reduction (all the quests in the game were already built using flags). 1056 variables, which were reduced to 780, were now reduced to 522 overall, with plenty of global variables available. A screenshot showing how Fall from Space utilized variables for tracking coins. With the majority of the game engine built, quests and coins fitting 16 variables into one, and no doubt other efficiency improvements I'll make along the way, the project in its full scope was viable again. ## Conclusion Variable counts are just as important

as file size in how feasible your GB Studio project is, but somewhat of a hidden issue. This needs to be taken into account for all projects, but particularly those larger in scope.

I'm still likely to optimise further, with all the hidden nooks and crannies I've snuck local variables into, but the project is back on track, after a (thankfully brief) struggle. Hopefully any other GB Studio devs who weren't aware of this hardcoded limit can take steps earlier than I did and save themselves the headache!

Thanks to Chris from Spacebot Interactive for tips on increasing the total available variables slightly, as well as how to get that combined total in the first place.

You can check out the demo of Fall from Space and follow its progress, over on my Itch page. Joel Jarman

Joel Jarman

Independent Game Developer (part-time), Game Boy Nerd & Graphic Designer (full-time).

www.joeljgames.co.uk ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### *Related*

VI. Hardware, ROMs & Emulation

- A Guide to SGB Mode
- Compiling GB Studio Roms into Standalone Executables
- Converting your GB Studio ROMs for use with the N64
- Distinguishing Reflashable Bootlegs
- GB Game Cartridge Shells
- Game Boy Camera+ Mod
- Game Boy Color Modes
- Getting GB Studio Games Running on the Xbox 360
- Getting Your GB Studio Games Running on the Wii U
- Getting our GB Studio games running on the Nintendo DSi
- Getting our GB Studio games running on the Nintendo DS
- Getting your GB Studio Games on Real Hardware
- How to Play a Game Boy Game on a Game Boy
- Playing GB Studio Games on a 3DS
- Playing GB Studio ROMs in Minecraft

- Porting GB Studio Games to Dreamcast
- Porting GB Studio Games to the GBA with GBSenpai
- Possibly the WORST way to play Gameboy Games
- Reflashing GBA Bootlegs with an NDS or NDS Lite
- Running My GB Studio Games on Xbox
- The Goomba Color Emulator

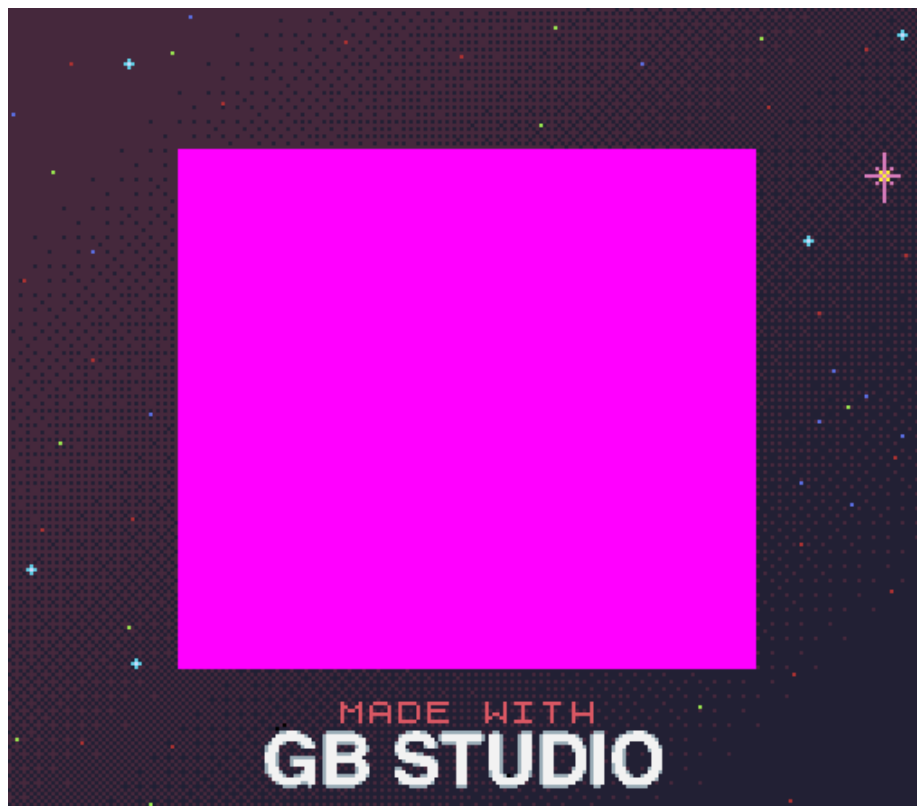
A Guide to SGB Mode

When the *Super Game Boy* was released in 1994, pretty much every Game Boy fan was ecstatic to try out their favorite handheld games on their big screen TV via the Super Nintendo peripheral. Fast forward a whopping twenty eight years and the SGB has been offered a new lease on life with the release of GB Studio 3.0 and the newly added *SGB mode* feature. During my time developing *In The Dark*, I made it my goal to figure out precisely how this admittedly strange yet wonderful mode worked.

In The Dark (2022) as seen in SGB Mode In this detailed guide, we will explore how the *Super Game Boy* functions and how to implement its many features in your own GBS3.0 projects. Right, let's SGBegin! ## **The SGB Border** One of the more exciting features the SGB offers is the full color border. Many Game Boy games back in the day used this border to great effect. Some used it as a simple aesthetic upgrade while others offered something more immersive, such as displaying the game within a simulated arcade machine.



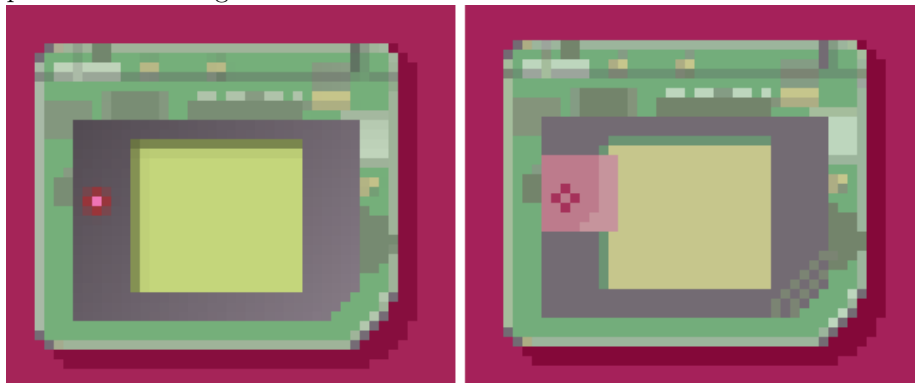
Donkey Kong Land 1995 (left) and *Donkey Kong* (1994) as seen when using the SGB Implementing a border in GBS3.0 is the easiest aspect of using SGB mode. Simply open a new project, go to **Settings** and click on the checkbox next to *Enable Super GB Mode*. Now make a build of your project and GBS3.0 will automatically create a 256 x 224 pixel *PNG* file. You can find this file (*border.png*) within the newly created *sgb* folder in your projects assets directory. The SGB will render this *PNG* file *as is*, so editing your border is as simple as opening it in your favorite image editing software and you're on your way!



The border.png file. Edit the graphics around the fluorescent pink area. The pink area itself represents the game play area. For the purposes of this article, I have edited the default border PNG to create a GB Studio Central themed border. It's best to *NOT* test the border using the inbuilt GBS emulator as the colors are very desaturated and do not represent what is seen when using an actual SGB. Instead, I recommend exporting a ROM and using the emulator 'BGB' , going to **options > system** and selecting *Super Gameboy*. Resetting the game will enable this mode.

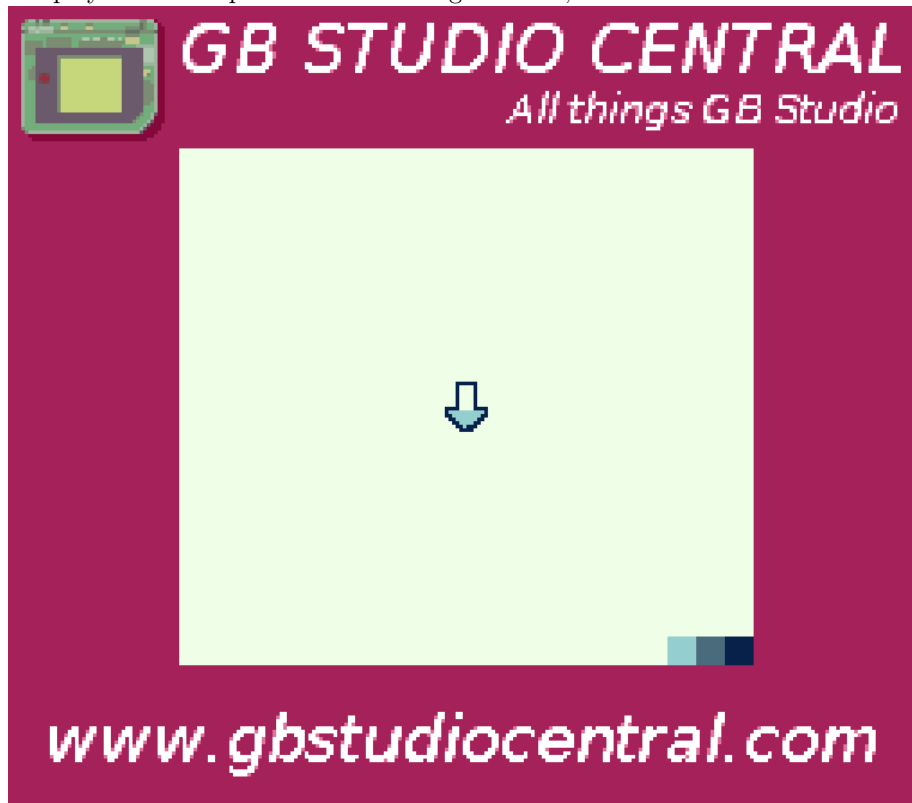


The edited *border.png* (left) and a screenshot of it running in BGB (right). Let's play a game of spot the difference with the two borders above. You will notice the red light on the Game Boy of the GBSC logo has not been accurately displayed in BGB. Developing games for the Game Boy always has its limitations so it should come as no surprise that creating a border for the SGB comes with its own considerations.



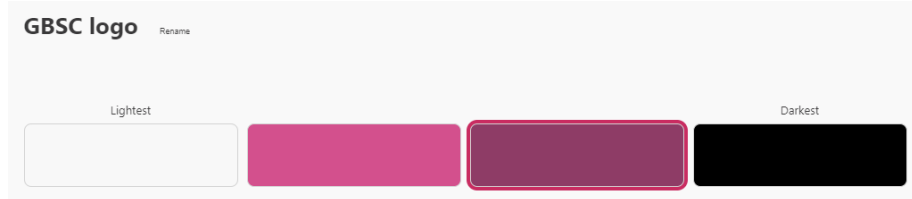
Graphical glitch owing to SGB hardware limitations (right). The large amount of anti-aliasing in *border.png* (especially the text) has pushed the unique colors well beyond what the SGB can manage. Specifically, there are two hardware limitations to keep in mind when editing your *border.png* file: * The border cannot utilize more than 16 colors. * The border cannot contain more than 256 unique 8×8 pixel tiles. To be clear, the SGB itself actually allows for 4 palettes of 16 colors each when rendering the border (just count the DK' 94 border colors and you will find there is well over 16 there), but we are restricted to using only one of those palettes in GB Studio – hence the 16 maximum colors. Now that we have our own SGB border, we can move onto the more complex features: coloring your game! But before I get into the “*how to*”, let's first discuss what is happening technically when the SGB displays color *in-game*.
How Color in SGB Mode Works While the SGB does offer color

options, it's not at all similar to the options provided when developing a Game Boy Color game. Instead, SGB color mode displays color *over the top of a game running in DMG mode*. You can think of it like this: *Color is displayed over the DMG Game Boy scene as if we were to cover the TV screen with colored cellophane*. Because of this, we must understand that a scrolling scene in a platformer or shmup for example, will have unintended effects when played on the SGB. It is for this reason that games with a static screen are best suited for SGB mode as zones of color are not attached to the scene itself but zones within the display of the TV screen. By default, background palette 5 of each scene is displayed as the color palette when played on the SGB. It is this palette that acts as the "cellophane" wrapped over the TV screen image. Furthermore, it doesn't matter what art element is beneath the cellophane so to speak, whether it's a background tile or a sprite, *all visual elements will be rendered with this palette on the SGB*. Returning to our screenshot in BGB (and with much of the anti-aliasing removed to bring us down to 16 unique colors), we can see that the placeholder scene in GBS3.0 has been colored with *default palette 5* (the fifth background palette in GBS3.0's default placeholder scene). If we were to set the player actor's palette to something different, it would not make a difference.

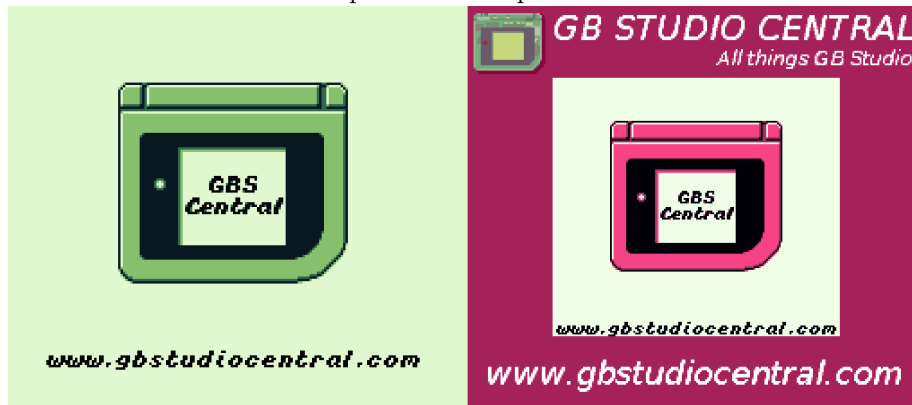


The placeholder scene as seen in BGB. The fifth background palette of this scene acts as the "cellophane cover" over the game view. ## Col-

oring a Scene in SGB Mode To start, let' s enable color mode in our project, create a new scene, set a newly created palette to the scenes palette number 5 slot and see what happens. Here is our newly created palette:



And a scene with the above palette set to palette number 5' s slot within it:



The raw PNG background (left) inserted into our scene and shown in BGB (right). If you look carefully, you can see that something is wrong with the color display in the image on the right. I have set the lightest tone in my palette to white, something BGB would have no trouble rendering correctly, and yet we have a very light green displayed in SGB mode in its place... something rather odd is happening here. The answer to this riddle is a strange quirk in how the SGB handles the first color slot when displaying colors. In fact, it' s pulling the color in slot one of palette 7 in the scene! If you want an answer to why this happens, it' s best you get in contact with the developers of the Super Game Boy itself but fact is fact – and it' s something we must work with. Here is the same scene but I have set our logo palette to palette 7 in addition to palette 5:



Our GBSC logo scene as seen in BGB. This time with our logo palette set to palette 7 too. It now displays white, the intended color. We have figured out the why and how' s of the SGB, but our journey into SGB mode doesn' t end here. Next, let' s take an in depth look into how we can color a scene with more than one palette! ## **Color Zones within Color Zones** To achieve more than one palette displayed at any one time in a scene, we are going to have to download the GB Studio Transfer SGB Packet Plugin by *NalaFala (Yousurname)*. To set it up, create a folder titled “*plugins*” in the root of your project directory and drag and drop the “*events*” folder found in the downloaded plugin inside. You can now click the “+ Add Event” button and search for “*Transfer SGB Packet*” to use this new script (close GB Studio and re-open it if required). Let' s take a look at it' s parameters:

▼ Transfer SGB Packet ▼

Command

Apply Palette to Rectangular Area ▼

☐ Inside surrounded area
 ☐ Surrounding character line
 ☐ Outside surrounded area

Top Left Position

X

0

Y

0

Bottom Right Position

X

0

Y

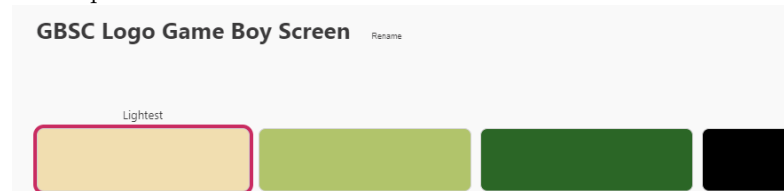
0

+ Add Event

The Transfer SGB Packet parameters Currently, this plugin only supports implementing a zone of color shaped like a rectangle but you can still achieve much with this. As the descriptions imply, clicking on any one or more of the checkboxes will enable you to color an area of the scene with a palette of your choice. It is important to note that the palettes available to us with this feature are limited to palette 5 to 8. The parameters below set the area of the rectangle using the x and y coordinates of the tiles within the scene and will *NOT* go beyond the maximum length and height of the Game Boy screen. In other words, the maximum value of x and y is 19 and 17 respectively (the width and height of the Game Boy screen itself when split into 8 x 8 pixel tiles). Let's create a new palette and set it to the 6th palette slot within our scene. In this way we can begin coloring the scene in more detail. In particular, we will color the Game Boy screen in the GBSC logo so it looks like the classic DMG colors, similar to the following:

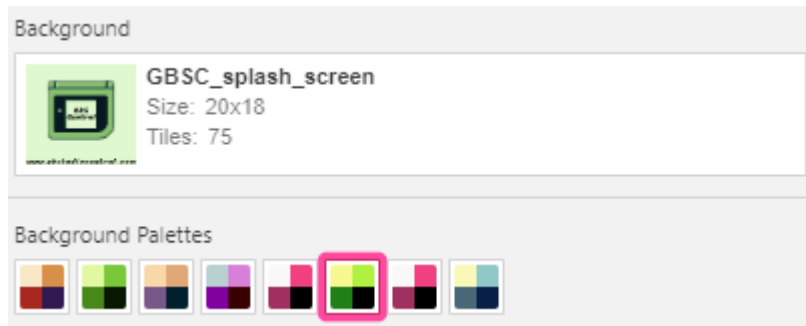


The GBSC logo scene as seen in GBS3.0 Here is the palette I have created in order



to accomplish the above when in SGB mode:

An approximation of the colors displayed on the DMG screen Setting this palette to the 6th slot in our scene, our Background Palette set now looks like this:



Palette 5 set to the logo palette, palette 7 set to the same palette to make sure the SGB renders the white color correctly and Palette 6 for our additional “Game Boy screen” color Now we can add a *Transfer SGB Packet* script in the *ON INIT* of our scene, making sure to create a rectangle that is colored using palette 6. In this case I have elected to color the rectangle inclusive of the x and y coordinate outlines by ticking the *surrounding character line* in addition to *Inside surrounding area*.

▼

Transfer SGB Packet

▼

Command

Apply Palette to Rectangular Area

☒ Inside surrounded area

BG Palette 6

☒ Surrounding character line

BG Palette 6

☐ Outside surrounded area

Top Left Position

X

7

Y

6

Bottom Right Position

X

12

Y

11

+ Add Event

The Transfer SGB Packet script Having exported a ROM and tested it in BGB,



the outcome is as follows:

The outcome as seen in BGB Hmm... again, we have a strange anomaly! While some of our green colors are now colored as intended, the yellow color of our dot matrix screen is a stark white. That's right, the first color of palette 7 is being called to color palette 6's first color too. This is the case *for all palettes used in SGB mode* and something that will need to be carefully considered during your planning phase. Knowing this, we can now redesign our background PNG to suit all these limitations. The end result is this:



The final scene with multiple color palettes displayed, all within the limitations encapsulated in SGB mode. I wonder what happens when you press 'A' ...

To summarize:

- * SGB mode lays color over the Game Boy view, it does not technically color a GB Studio scene
- * The SGB Border must use a maximum of 16 colors
- * The SGB Border must contain a maximum of 256 unique tiles
- * By default, the palette 5 slot in any one scene is used to color that scene
- * Palettes 1 to 4 in any one scene *cannot* be used when in SGB mode
- * Palette 7's first color slot will always render over any other palettes first color slot
- * All visual elements will be rendered with palettes 5 to 8 on the SGB, irregardless of actor palette settings

Having read this rather long guide on how SGB mode works, you might be thinking that there are a lot of weird quirks to consider when creating scenes for the *Super Game Boy*. And you're not wrong. But at the same time, exploring this niche aspect of Game Boy development can be a rewarding experience for anyone who cares to walk this rarely trekked road. I certainly enjoyed implementing this feature when creating *In The Dark*, despite all the limitations. Ultimately, *SGB Mode* is a wonderful extension to the already heavily limited process of Game Boy game development, and a road well worth traveling! If you want to check out the final build of the project described in this article, see how I got around the palette 7 issue (and find out what happens when you press A), the source code and ROM can be found [here](http://www.gbstudiocentral.com).

Thanks to *NalaFala (Yousurname)* for their help in the creation of this article.
Gumpy Function

Gumpy Function

Independent Games Designer, Artist, Film Enthusiast and Full-time Dad (he/him). Check out my games here!

www.gumpyfunction.com/ ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### *Related*

Compiling GB Studio Roms into Standalone Executables

Recently, a way was discovered to compile the web exported versions of a GB Studio ROM into a stand alone executable (Windows or macOS). Thanks to the GB Studio Discord user MrPlayerYork for releasing this info on his Youtube Channel (video linked below). Some of the things outlined in his video are a bit unclear, so I am going to go through the process and detail some of the issues I had along with the solutions. # Before You Begin This tutorial expects you to have some comfort or knowledge of using a Terminal or Command Line Interface and a Text Editor. Everything is laid out step by step, so even if this is your first time using those tools, you should be able to follow through it. Regardless of which platform you build with/for, you will need Node.js 14.15.4 which can be found at <https://nodejs.org/en> *Note: Older versions of Windows can install older version of Node.js, and it should still work, though some functions like compiling a dev/test build may not.* If you are on Windows, you can download and install Visual Studio Code at <https://code.visualstudio.com/> for an easier workflow. This process can also be done with any text editor (Notepad on Windows or TextEdit on macOS for example) and simply the Terminal or Command Line Interface if you choose. We haven't been able to use this method to successfully cross-compile (use Windows to build for Mac and vice-versa), but there may be some dependencies or a step we are missing. # Export Your Game First we need to export your game for the web. This is easy, in the menu bar go to Game>Export as>Export Web. GB Studio will compile the game and open a folder. We will need this later, so keep this folder open for now. # Folder Structure Folder structure is important for your team's sanity, even if it's only you. While it's not necessary, it's highly recommended to follow this file structure and create these folders: **Note: Your folder names should not have spaces.*

ProjectName:

```

> builds
> src
  |-> assets
  |-> game

```

Copy the contents of the newly compiled web build folder into the */Project name/src/game* folder. That's it for this step. # Build Parameters If this is your first time compiling software, this could seem a bit daunting. If you take it slow and follow step by step you there shouldn't be a problem. # Windows Load up VS Code and open up the project folder. Go to Terminal>New Terminal, your window should look something like this: In Terminal, input this command: **npm init -y** You should see some alerts appear in the console, these are perfectly fine and nothing to worry about. Once it's done, you will notice on the left hand side a file named "package.json" appeared. Next in Terminal enter: **cd ./src** Then: **npm init -y** This will make second "package.json" appear in the src folder. Next move up a level to your project folder by typing: **cd ..** When finished, it should look like this: Next we need to change some information in the 2 newly created "package.json" files. This can be opened directly in VS Code by clicking on them. If you see a yellow line under your project name, that's because you used capital letters in its name. You can fix this by simply changing your project name to all lowercase. Go back to the terminal and enter: **npm i nwjs-builder -D** This step could take some time depending on your internet connection. You will notice that a few new things will have been added to your project: a file called "node-modules", a file called "package-lock.json", and the dependencies part of the "package.json". It should now look something like this: Next up we are going to make some more edits to the "package.json" in the root of the project folder. Delete this entire line: **"test": "echo \"Error: no test specified\" && exit 1"** And replace it with these 2 lines: **"dev": "nw src/ --remote-debugging-port=9222", "build": "nwbuild --platforms win32,win64 -o builds/ src/"** Don't forget to save the file when you have made the edits, it should look something like this when you are done. Next up we are going to open the "package.json" inside the src folder, the first change we are going to make is to this line: **"main": "index.js"**, We don't have this file so we want to direct it to our web build index.html file in the game folder. Do so by changing it to this: **"main": "game/index.html"**, Next up we are going to set the window information for our application to the "package.json" file under the scripts section, this is what we are going to add (make sure to copy all the text):

```

"window" :{
  "title" : "Sample Project" ,
  "toolbar" : true,
  "fullscreen" : false,
  "frame" : true,
  "width" : 320,
  "height" : 288,
  "max_width" : 480,

```

```

    "max_height" : 432,
    "min_width" : 160,
    "min_height" : 144,
    "resizable" : true,
    "position" : "center"
  },

```

Any of these can be changed to suit your needs there are also plenty of other options available on the nw builder github. Now you should be pretty much ready to test the program. You can do this by running this in your terminal. **npm run dev** You should be greeted with your game in a window. Here you can play around with the window settings listed above and see them in action before building your ROM into an EXE. When you are ready and happy with your exe, build your game by simply typing this command: **npm run build** This will output all your builds into the build folder of your project. # Resource Hacker The next step is to customise your app and make it your own. We are going to do this on Windows with Resource Hacker found here: <http://angusj.com/resourcehacker/> Open your exe in resource hacker to make some changes. I' m going to change my icon with a random image I found on the internet, but you can create icon files in gimp amongst other image editing software. When your exe is loaded into Resource Hacker, go down to "IDR_MAINFRAME" . Right-click on it and select the replace icon action. There are three files here, but I believe only the first is used. Next up we want to change the version info as it' s currently representing of node js not our app. Go to version, click on the only file in that category. Change this info to your liking, but maintain the structure to keep Windows happy. Hit save and your exe will change accordingly. # Building an installer If you would like to make an installer *Inno Setup* is recommended, you can find it here: <https://jrsoftware.org/isdl.php#stable> Follow their wizard to input all your info, select your exe and the folder that contains your build files and it will build a setup script. Go to **Build>Compile** and it will make your setup for you. You can then find the installer in the output folder and also under the build tab. All that' s left is to add that to the platform you wish to distribute your game on, whether it' s github, itch or maybe even steam. Original video by MrPlayerYork # macOS Building a standalone app from your web build is somewhat easier depending on how in depth you go. This is adapted from a video made by Sparckman. The same prep is required as before, make sure you have your GB Studio Web Build already exported and Node.js installed on your computer. This also only seems to work when building on a Mac, so you will need access to one in order to do this. Go to the web folder where your index.html file is stored. Create a file named "package.json" . Paste the following code into the "package.json" file and save it:

```

{
  "name": "nw-builder",
  "version": "3.5.7",
  "description": "nw-builder",
  "main": "index.html",

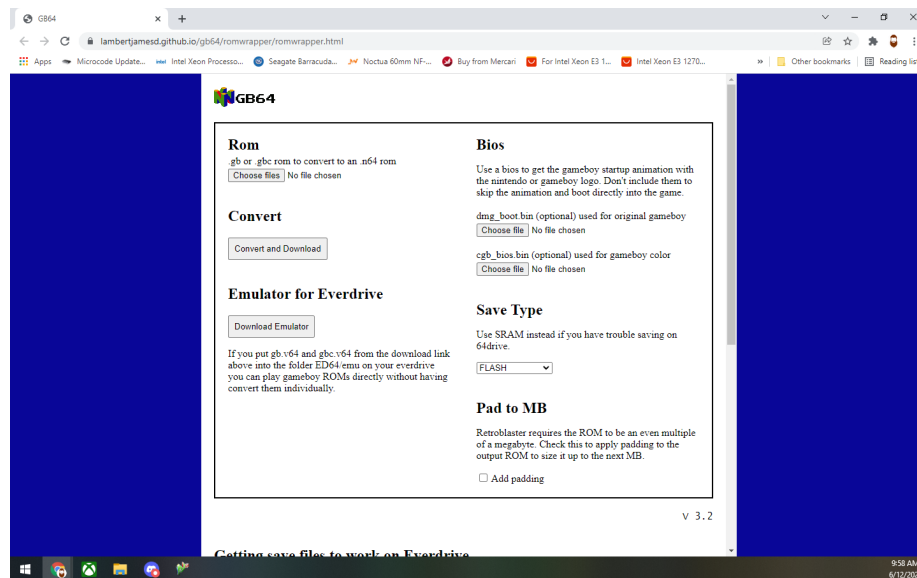
```

```
"dependencies": {  
}  
  
}
```

For these next steps, you will need to open the Terminal app and use a command called “sudo” to force the command to run as an administrator. You may get a warning to only use this command if you know what you are doing, but these commands shouldn’t cause a problem. Still, we need to warn that running commands as sudo shouldn’t be done without knowing what you’re doing. The first command installs the needed tools within node: **sudo npm install nw-builder -g** Next, change your directory to where your index.html file is. You can do this by typing in “cd “, space, dragging and dropping the folder and pressing enter. Paste the following command into Terminal (don’t forget the period at the end): **sudo nwbuild -n "Your_Game_Name" -p osx64 .** And that’s it! You should see a build folder appear with your game as an app and few other files. You may still want to tweak the package.json like listed above to play with the window settings, and you’ll probably also want to make a custom icon. Icons on a Mac are built from 1024×1024 png files, if you have one created you can follow this tutorial to convert into an *icns* file. You can then go to File>Get Info on both your icon file and your app, and copy and paste the icon in the top left of each info window.

Converting your GB Studio ROMs for use with the N64

So far, we have explored ways to have your GB Studio ROMs playing on the Dreamcast, Game Boy Advance and 3DS. Each time I discover a way to have our games running on another console, I have to let everyone know about it and now we can add the Nintendo 64 to the list! This actually might be the easiest conversion of a ROM ever. Take your ROM of choice, then go to this website. (I’ll be using the Jurassic Snap DEMO for the purposes of this article – I mean, it’s the perfect ROM for this.)



GB64 ROM conversion website. This is all pretty straight forward. Upload your ROM, then change the save type to *SRAM*. Ticking the “Pad to MB” option is also recommended. ****IMPORTANT* Adding a bios file to your ROM and distributing it is illegal and only adds the Game Boy boot logo on startup. If it's for personal use this is fine, as long as you legally source your bios file!*** Then click convert and download and you’re done! Unfortunately I was unable to get this working on any emulator but it was fully functional on my actual N64.



Jurassic Snap running on the N64. There is also a way to make physical N64 carts of your ROMs! I mentioned the RetroBlaster Programmer by RetroStage in an earlier article. This device can also flash N64 carts made by RetroStage. In a reasonable amount of time we have managed to get GB Studio ROMs working on so many different consoles it's amazing! I have very little doubt that our Game Boy games could be available for every console capable of running them before we know it. Veund

Veund

Game Boy Enthusiast, Hardware Fanatic, Mad Man. (he/him) ### Share this:

- [Click to share on Bluesky](#) (Opens in new window) Bluesky
- [Click to share on X](#) (Opens in new window) X
- [Click to share on Facebook](#) (Opens in new window) Facebook
- [Click to share on Mastodon](#) (Opens in new window) Mastodon
- [Click to share on LinkedIn](#) (Opens in new window) LinkedIn
- [Click to share on Reddit](#) (Opens in new window) Reddit ### Related

Distinguishing Reflashable Bootlegs

Some bootleg games can be reprogrammed with your own ROMs. But how can you tell if you have one of these hidden gems, and is it worth your time and

effort? Do note this method will require a screwdriver, but what type may vary from bootleg to bootleg. It should be one of 3 types: philips head, tri wing or game bit like the official carts. This will be up to you to figure out, your best option is to get a screwdriver set with multiple heads. There are many options online for these sets. You will also need a flasher that is compatible with flashing bootlegs. I mentioned a few in my previous article. # Step 1: Opening The Cart The first thing you need to do is open your bootleg. A number of different revisions have been released over time, and the best way to identify which one you have is to open it up. All you need to do is remove the tiny screw from the back of the cart, then slide the front part of the shell down so it can be pulled towards you. A Pokémon Crystal Bootleg

A Pokémon Crystal Bootleg A Game Boy cartridge opened with the printed circuit board exposed.

How the cart looks when opened. # Step 2: Identifying The Cart Now that the cart is open, you need to work out which cart you have. How can you tell if it's flashable? Luckily, someone has done all the hard work for us! Over at <https://flashcartdb.com/> there is a curated list of reflashable bootlegs. In the contents at the top of the page, click the link for GameBoy Clone/Repo Flash Carts. This will give you a table with pictures of the boards of a bunch of carts you can compare to your cart. Some of The Joey JR information may be outdated, check out the BennVenn discord for more info. In my case it was a M29W640 SD007 48BALL-64M. It is reflashable but has no battery or holder, so saving won't work unless one is installed. https://flashcartdb.com/index.php/M29W640_SD007_48BALL-64M # What To Do Now? The first time you plugin your bootleg, you should make a backup of the ROM. This is not crazy important, but some have modified roms to handle batteryless saves and they could be useful sometime in the future or if you ever want to revert back to the original ROM. Now you are ready to reflash your bootleg with your GB Studio game. These bootlegs don't have any write protection so they can be reflashed as many times as the flash chip can handle so don't worry about having to reflash if you need to update your game. # Final Notes Bootlegs may vary in size. A lot of them are either 2 MB or 4 MB – this is massive in size for a Game Boy game. You will be hard pressed to try and fill all that space with a GB Studio ROM, and the max size a GB Studio Game can be is 4 MB. These bootlegs are perfect for personal use with GB Studio games. I myself have about 20 of them I use for multiple games and for \$3 each you really can't go wrong. Veund

Veund

Game Boy Enthusiast, Hardware Fanatic, Mad Man. (he/him) ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon

- Click to share on LinkedIn (Opens in new window) [LinkedIn](#)
 - Click to share on Reddit (Opens in new window) [Reddit](#) [### Related](#)
-

Game Boy Camera+ Mod

Released in 1998, the Game Boy Camera (known as the Pocket Camera in Japan) quickly became popular, even being used for the cover of Neil Young’ s Silver & Gold album. The photos it captured were tiny (0.014 megapixels, or 128×112 px), and so was the hardware — for the time. The unique, barely pocketable cart shell featured a bulbous, swiveling lens housing and a fixed, wide-angle lens. This lens choice limited creative options, reducing objects just a few steps away to a handful of pixels. It only took two decades and change to reimagine the Game Boy Camera, but here we are: the Game Boy Camera+ mod by 2BitToy recently hit the retro gaming community, promising improvements to both form and function. Was it worth the wait? Let’ s find out.

(Game Boy Camera and Pocket Camera)

Screenshot from GB Studio using photos shot with a modded Game Boy Camera+. Photos by Martin Cremerius-Gauer.

(Game Boy Color with Camera+ mod and 5-50mm lens)

Camera+ with a 5-50 mm zoom lens and 3D printed lens sleeve on a FPGB console ## What is the Game Boy Camera+ mod? The Game Boy Camera+ mod is a 3D-printable shell for the stock Game Boy Camera PCB (as well as HDR’ s Game Boy Camera PCB) that redefines the original shell’ s shape for a straightforward experience, and enables the use of CS-mount lenses. It’ s also compatible with the Game Boy Camera Flashcart PCB that was designed by HDR. This mod is fully reversible and non-destructive, meaning you can revert the Game Boy Camera to its original form at any time. No soldering required! ## Here’ s what you need You’ ll need several components to start building your custom Game Boy Camera shell, in addition to the Game Boy Camera itself. ### STL Files There are four STL files that make up one Camera+ Mod. A front part, a back part, the sensor mount and an optional filter holder. The STL files (3D printable files) can be bought at 2BitBoy’ s store, including three versions of the shell: one for the stock GB Camera with a 30mm CS mount ring, one for a 31mm CS mount ring and a separate version for HDR’ s custom GB camera PCB. ### CS-Mount Ring Adaptor For a CS-mount lens to be screwed onto the shell, you’ ll need a 5mm C-CS mount adaptor ring. These usually cost around five dollars and can easily be found online. They come in 30mm or 31mm diameter, which must match the 3D printed shell you chose above.

CS-mount ring adaptor ### 3D printing / ordering If you have a 3D printer of your own, then you can print the shell yourself. I got good results with PETG and fine print settings. If you don’ t have a 3D printer, then you can order a print at a 3D print service like Shapeways. I recommend ordering it in MJF

(Nylon), which is way more durable than PETG and has a very lovely look that makes it almost feel like professional equipment. I got myself a set for under 15 bucks at JLC3DP. Just upload your STL files and make sure to select MJF (Nylon). Leave the rest as is, and order the parts you need.

Different 3D print materials. All are printed in MJF Nylon, except the far right one, which is printed in PETG ### Filter If you feel fancy and want the full experience, then you can get a 10mm diameter IR-cut filter that will sit between the sensor and the camera lens, mostly for protection.

(Camera+ IR filter)

Infrared filter inserted in the Camera+ shell, protecting the Game Boy Camera sensor Once you have all the parts, you can assemble them. Watch Macho Nacho's YouTube video for a detailed build guide. It's straightforward, and the only tools you need are a tri-wing screwdriver to open the original Game Boy Camera and a strong, thick glue. I recommend a five-minute 2K epoxy adhesive. ## Lenses As mentioned before, the Camera+ mod uses CS-mount lenses, which are mostly found in security cameras. Therefore, there are a handful of quite good lenses on the market for under 20 dollars. I own a 6-60mm and a 5-50mm lens, both cheap ones from Aliexpress. The 5-50mm lens seems to be a very good all-rounder, as it's noticeably smaller and lighter than the 6-60mm lens. You can also 3D print (or order) a specially made sleeve for the 5-50mm lens, which I printed myself in PETG, that lets the lens look and feel like a big boy camera lens.

(5-50mm zoom lens with sleeve / 6-60mm lens without sleeve) ## How to get the photos off the Game Boy Camera There are multiple ways of getting photos taken with a Game Boy Camera onto a computer or smartphone. Let's quickly go over the two most common options. ### GB Operator or GBxCart RW Devices like the Epilogue GB Operator or the InsideGadgets GBxCart RW can read and write data such as saved games back and forth between Game Boy cartridges and your PC. They can also be used to get the photos from a Game Boy Camera. Just plug the camera cartridge in and save the images to your computer.

(Game Boy Pocket Camera inserted into a GB Operator) ### Wifi Printers Another option is a Wi-Fi printer, which is a small device that you plug into the Game Boy / Color link cable port and (most often) power via a 5V USB connection. You would "print" your photos of choice to the Wi-Fi printer, which, to the Game Boy/Color, is just a normal Game Boy printer. But instead of printing the images out, it saves them on the device itself. Open a Wi-Fi hotspot, go to a specific IP address or URL, and then download all the photos directly through the browser (no internet required).

Wi-Fi Printer module connected to a Game Boy Color ## Game Boy Printer If you prefer to see your pixels on paper, you can, of course, just print your photos with the original Game Boy Printer.

Game Boy Printer As it's the nature of photography, this is only the entry of the rabbit hole that is Game Boy photography. There are different Game Boy Camera PCBs to explore, which let you upload custom camera ROMs with more features, lots of different lenses to try, use big lenses to take pictures of the moon, take color photos with the Game Boy using filters and so much more! Martin Cremerius-Gauer

Martin Cremerius-Gauer

Martin first went into making GB Studio games with his playable portfolio of his own personal website. He is generally interested in a very broad band of topics, and adds new things to the skill list all the time. Being born in the late 80s, the original Game Boy DMG-01 was his first gaming console, and so he kind of has a nostalgic romance with everything around it.

2ndlawnames.com ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### *Related*

Getting GB Studio Games Running on the Xbox 360

I have wanted a modded Xbox 360 for ages, and earlier this year I picked up an RGH2 phat Xbox 360, so you guessed it, I worked out how to get our GB Studio games working on the system! Modding your Xbox 360 is beyond the scope of this article and is honestly a rabbit hole to dive down. Much like the original Xbox there are many different setups, I'm running the Aurora dashboard on a Falcon revision Xbox 360 using RGH2. An image of the XBOX 360 user interface, showing multiple games. I found there was an emulator called GameBoy 360, but after some looking into it this was built off Retroarch 0.9.8.0 (and the latest Retroarch is 1.0.0.2), so I opted to grab Retroarch over GameBoy 360. Oddly enough this wasn't found on the Retroarch website like all other versions, this was found on a Digiex forum here. An image of the XBOX 360 user interface. The GameBoy 360 emulator is shown. An image of the XBOX 360 user interface. The Retro Arch 360 emulator is shown. My Xbox 360 came preconfigured, so all I had to do was copy retroarch to \Hdd1\Emulators and it appeared on Aurora's menu screen. You may need to add emulators as a path under content in Aurora (mine specifically has a scan depth of 3).

Finally you need to add your GB Studio game. This has to go inside the roms folder in Retroarch. You can add as many sub folders as you like (I put mine inside Gameboy). Once you have done that, load content, then navigate and launch your game; oddly enough the B button is enter and A button is return.

I' m not sure why this is the case but let' s roll with it. An image showing gameboy content being loaded in the emulator. A screenshot of the Mother Demake title screen. Your game should launch with no further action needed. The Xbox 360 has a still-thriving homebrew community. This marks another console that we can run our GB Studio games on! Veund

Veund

Game Boy Enthusiast, Hardware Fanatic, Mad Man. (he/him) ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
 - Click to share on X (Opens in new window) X
 - Click to share on Facebook (Opens in new window) Facebook
 - Click to share on Mastodon (Opens in new window) Mastodon
 - Click to share on LinkedIn (Opens in new window) LinkedIn
 - Click to share on Reddit (Opens in new window) Reddit ### *Related*
-

Getting our GB Studio games running on the Nintendo DSi

At this point, it shouldn' t be too much of a surprise that I enjoy getting our games working on devices that they aren' t really supposed to be running on. Recently I picked up a DSi, and the first thing I did was get our games running on it. This will require a modified DSi; modding the console is beyond the scope of this article but you can follow the instructions here. ## **Twilight Menu** ++ Twilight Menu ++ is the new replacement launcher typically installed when you mod a DSi. What is great about Twilight Menu ++ is that it contains everything you need to run various roms out of the box, which includes the Gameboy. You simply need to copy the GB Studio roms to the SD card of the DSi, then launch them in the console. Twilight menu running on a Nintendo DSi. This had to be the easiest experience I had at running any emulated Gameboy game ever and highly recommend breaking out your old DSi and doing so if you happen to have one. Running a gameboy game on a nintendo DSi. ## Bonus Twilight Menu ++ also has the capability of running from Nintendo DS flash carts and Nintendo 3DS consoles by running the .nds file on a NDS flash cart or installing the .cia file on a 3DS via FBI, and can be run in widescreen on the 3ds using patches made with TW Patcher available from universal updater. Twilight Menu running on a Nintendo 3DS XL. Veund

Veund

Game Boy Enthusiast, Hardware Fanatic, Mad Man. (he/him) ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn

- [Click to share on Reddit \(Opens in new window\)](#) [Reddit ### Related](#)

How to Play a Game Boy Game on a Game Boy

With today’ s modern gaming conventions, the original Game Boy can be a mysterious and confusing device for today’ s player. That’ s why we’ ve created this guide to make sure you’ re able to get the most out of your Game Boy. ## Step 1: Get a Game Boy Here you can see my modded Game Boy. It’ s awesome. Feast on it’ s glory. A Cherry Red Game Boy Color with a large IPS display.

This beaut has a large IPS display installed by Veund While this is technically a Game Boy Color (Cherry Model), older DMG and Pocket-style Game Boy’ s are suitable. Different colored shells are also acceptable. On the back of the Game Boy, towards the top, you will see an empty void. This cut-out, known as the “cartridge slot” , is where you place your game. Looking down the top of a Game Boy with no cartridge installed to see the empty cartridge slot.

The “cartridge slot” ## Step 2: Get a Physical Game While you are able to download games on modern consoles, the Game Boy was created before the Internet was available in most homes, leading Nintendo to decide to release games on physical cartridges (colloquially known as “carts”). You’ ll need to get one of these. While there are plenty of secondhand markets, we suggest looking at recent homebrew games to support independent Game Boy developers. *Note: The Game Boy is only designed to work with Game Boy Games. You will not be able to use other games with your Game Boy. As you can see below, this PokéROM won’ t work.* A mini CD labeled “Mew 151 PokéROM PC/Mac Compatible” . It is placed slightly into the slot of a clear purple Game Boy Color. A label on the back of the Game Boy shows an image of a Charizard Pokémon and is labled “Game Boy Color Light” .

Using a Mew Game Boy still wouldn’ t suffice.

Photo courtesy of Veund Here we have a signed copy of *Deadeus* (thanks IZMA!) still sealed in its box. Unfortunately this game won’ t work since the box won’ t fit in the cartridge slot and opening the box would devalue it significantly. A boxed copy of a Game Boy game attempted to be placed into the empty cartridge slot, which it won’ t fit in as the box is bigger than the Game Boy. The art on the box displays what looks like an intestine with the game’ s title, “Deadeus” , overlayed. The publisher’ s name, “Spacebot” , is written on the side of the box.

Sadly, you need to open the box to make this work Note other included items with special editions also won’ t work. As you can see here, the audio cassette OST included in the deluxe edition of *Deadeus* also doesn’ t fit the cartridge slot. A sealed audio cassette for the soundtrack to the Game Boy Deadeus attempting to be placed in the empty cartridge slot of a Game Boy.

This cassette doesn't work in my 8-track player either. Luckily I kept my library of older Game Boy games from when I was younger, so I have an original release copy of *Shadowgate Classic*. I'm happy they published this one after the public backlash over the reformulated *New Shadowgate*. An open cartridge cover for a Game Boy game next to a Game Boy. Inside is a black cart with the label for *Shadowgate Classic*.

A classic indeed. ## Step 3: Put the Game into the Cart Slot The cart fits nicely in the cartridge slot, provided it is oriented correctly. The label should face out towards you as you look at the back of the Game Boy. The "teeth" at the bottom of the cart (these are officially known as "pins") are designed to make contact with connectors in the Game Boy's slot facilitating the transfer of data between the console and the chips contained within the cart. Luckily, the shape of both the cart and the slot are designed to prevent you from inserting the cart in the wrong orientation. A Game Boy cart placed incorrectly into the Game Boy. The label is facing the wrong way and the cart is angled showing that it will not accept the cart in this orientation.

The wrong orientation. Notice how the label is not visible? A Game Boy game correctly inserted into a Game Boy.

The correct orientation. The label is visible, and the slot fully accepts the cart so that the top of the cart is flush with the Game Boy. ## Step 4: Insert Batteries Unlike modern handhelds that have fancy embedded rechargeable lithium-ion batteries, this old boy needs removable ones. If you turn on your Game Boy and nothing happens, the first thing to check is if you need fresh batteries (either newly purchased or fully charged if using rechargeables), or if you have any batteries installed at all. Different models of Game Boy will require different types and number of batteries. You can access the battery port by unclasping and opening the cover on the back. If your Game Boy is not modified or re-shelled, chances are it has no battery cover at all, and you can easily see whether batteries are installed. This Game Boy Color requires two AA batteries. A Game Boy with the battery cover removed. There are no batteries installed.

Can you see the problem? A Game Boy with 2 AA batteries installed. A hand giving the "thumbs up" gesture is in front.

Problem solved! ## Step 5: Turn the Game Boy "On" Modern consoles have "standby" features and buttons to wake the device, but the Game Boy will need you to physically move a switch into the "On" position. Think of it like a light switch supplying power to a light bulb. Depending on your model of Game Boy, you might find this switch on the top of the Game Boy. On my Game Boy Color, it's located on the side. The side of a Game Boy Color. A grey switch is located in the middle. It is currently in the "Off" position.

Push this sliding switch "up" to the "ON" position. ## Step 6: Enjoy Your Game You now have everything in place to enjoy your Game Boy game. Play through and experience the glory that is this handheld 8-bit wonder. A Game Boy turned on. The title screen for *Shadowgate Classic* is displayed. A language

selection option is available. While there's nothing like playing a Game Boy game on hardware, you're not out of luck if you don't own one. Thanks to dedicated individuals in the homebrew and modding scene, you can play Game Boy games on a myriad of devices including the PS3, WiiU and Original XBox. Fortunately, we have the wonderful work of Veund that details how to get this done, check out the many articles he's contributed to our site here: <https://gbstudiocentral.com//author/veund/> Emi Paternostro

Emi Paternostro

Audio Engineer, Mac Technologist and Video Game Developer. Managing Editor of GBStudio Central. (he/him)

proximitysound.com ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### *Related*

Playing GB Studio ROMs in Minecraft

As many of you know, I love to get our GB Studio ROMs running on various devices such as the Original Xbox or iPod Nano, but recently I picked up Minecraft for the 1 millionth time and asked the question, "Can I play Game Boy games in Minecraft?" – and it turns out you can! A screenshot from Minecraft, showing a large rustic house. # Requirements You will require the ability to host a server and run a modified version of the Minecraft server known as Spigot. Spigot allows you to run server-side plugins. This is great because people won't be required to use a modified client; they just connect to your server and they can play any of our ROMs. The only plugin you need is Game Boy. You can also use ProtocolLib, which will give you a smoother experience, just put these in your servers plugins folder (if your server hasn't made this folder just launch it then close it after spawn is created and it should be there). An example image showing a list of files in the server's plugins folder. # Adding ROMs Inside the plugins folder, there should be a Game Boy folder. If there isn't one, launch the server with the plugins installed, then shut down the server. Inside the Game Boy folder you'll find a ROMs folder, just put your ROMs in there and launch the server. An example showing the roms folder selected. An example of a rom file; the selected game is the Pokemon Scarlet Version demake. # Playing our ROMs Now that your roms are installed on the server and you have connected your client, open chat and type `"/gb"`, it will list all the games loaded on the server. Open the chat menu again and click the rom you would like to play, and it will give you a map where the Game appears on. A screenshot from Minecraft

showing a map. Another screenshot from Minecraft. Now, the map is displaying the Gameboy rom in action! Close text chat. The controls are a bit odd but you get used to them pretty quickly.

Jump-Key = A button
Right-Mouse-Click = B button
Swap-Item-Key = Start Button
Q-Key = Select Button
WASD = Arrow keys

It works pretty well, my server crashes every now and then but that could be due to another reason as this isn't the only plugin I use. Also saves are tied to the user not the server, so you don't have to worry about someone else overwriting your progress.

This would be an awesome way to show off your games, build a massive impressive structure, or interactive mural around your game, then allow people to play it inside Minecraft. I would be pumped to see people do that. Veund

Veund

Game Boy Enthusiast, Hardware Fanatic, Mad Man. (he/him) ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### Related

Porting GB Studio Games to Dreamcast

On the 8th of September, Ian Michael posted a video of his version of GnuBoy running on the Dreamcast. This version auto boots a supplied rom and allows you to make custom splash screen images, making it perfect for our GB Studio games.

GB Stadium running in the ReDream Emulator So how exactly do you get this working with your games? For those unfamiliar with the Dreamcast homebrew scene, the Dreamcast-talk forum linked in the video description doesn't explain all the ins and outs too well. Luckily, this is actually easier than it first appears. When you scroll to the bottom of the Dreamcast-talk forum post you will see that 4 files have been uploaded. Specifically, you will need *gnuboy[Auto]plain[fix]s.rar* and optionally *adxencd[converter].rar*.

Attachments found in the Dreamcast-talk Forum (Note: these files are subject to change as they are further updated)

Next you will need to get *BootDreams*. This gave me security warnings on my PC, so if you aren't comfortable using it stop here. BootDreams is a widely used program in the Dreamcast community, so I think it should be fine to use but if you are worried I took the risk so you don't have to. Next up, unzip *gnuboy[Auto]plain[fix].rar*. Here you will see a number of files and folders. Go into the folder marked *gb*, there you will see *gbs.gbc*. Replace this gbc rom with your desired game (make sure you keep the same file name). For the most part, this is all you need to do, but it wouldn't suit your game without changing some of the images in the folders *INTRO* and *pics*. The order of the files that are displayed on screen are as follows: * P.png * P2.png * adx.png * RAD.png * RAD2.png * imrok.png * L.png * l2.png For the intro, you can remove some of these images but it will hang on a black screen until it loads. The next image that opens before the rom is loaded is *startup.bmp* in the *pics* folder. All images need to be 640×480 and compressed in order for them to work correctly. You can use this website to compress them. Here is where the optional download comes in handy. If you want to change the audio used in the intro, you can do so by following these steps: * Unzip *adxencd[converter].rar* * Find/create a .wma audio file * Place it into the same folder as *adxencd.exe* * Run *adxencd.exe* to convert the audio file to the appropriate format * Overwrite the current audio files in the *INTRO* folder So now your GnuBoy should be set up for your GB Studio game. All you need to do now is install and run BootDreams.

The BootDreams Interface Select the root folder of *gnuboy[Auto]plain[fix].rar*, change the CD Label to the title of your game, leave the next field as *Audio\Data* and click *process*. BootDreams will make a .cdi file which you can name as you wish. Then burn this .cdi file to a CD at the lowest writing speed you can. Finally, put the CD in your dreamcast and off you go – you have successfully ported your GB Studio ROM to the dreamcast! This is still early on in development, there are more features planned such as full CD quality audio which is very exciting! I can't wait! As a lot of people are swapping out the Dreamcast optical drive assembly for the GDEMU, you may want to also release the CDI image if you plan on doing a physical release. It's so awesome that we can add Dreamcast to the devices that GB Studio games can be released on. Veund

Veund

Game Boy Enthusiast, Hardware Fanatic, Mad Man. (he/him) ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### Related

Possibly the WORST way to play Gameboy Games

For those who don't know, there are 2 things I love to do: repurpose old hardware, and get our GB studio roms running on things that weren't designed to be run on. So when i stumbled across this Youtube short from JayBoyModz, I had to put it to the test and try playing our roms on the ipod Nano. An image showing the Rockbox user interface running on a silver iPod nano. This works by installing a custom firmware onto the ipod called Rockbox. It is compatible with a number of different music devices listed on the website, but not compatible with every ipod so if you want to try it, check the website first. Rockbox includes a Gameboy emulator called Emuboy, a port of GnuBoy built specifically for Rockbox.

So what makes this the worst way to play Gameboy games? First you might have noticed from the pictures, but the ipod nano has a screen resolution of 176 x 132 while the Gameboy has a screen resolution of 160 x 144. This means that some horizontal lines are lost to fit the game on the screen, which can cause some GB Studio text to become unreadable. The homebrew GameBoy game Stardiver running on the iPod nano. In addition to the screen issue, there are some hardware problems. The Gameboy has 8 buttons while the ipod has 1 button and a scroll wheel with some directional buttons underneath, giving you 5 inputs. This means that 3 of the inputs are mapped to swipes of the scroll wheel, which is a bit messy for games that require quick, precise input.

The ipod nano also doesn't have the power to emulate Gameboy very well, requiring some frameskip to get the games running well enough to play. This is fine for me, but if you are susceptible to choppiness or frame tearing this could be a deal breaker. This could be different on different music devices.

I also experienced this odd input problem where I would use the A input on my ipod and B would also register; I'm not sure if this was an issue with my ipod nano or an issue with the emulator. This problem persisted even after changing the B input in the menu, but button mapping is not a fully implemented feature of the emulator yet.

All in all this was a fun experiment to try, but I wouldn't recommend doing this as a daily driver dedicated Gameboy device as just about any handheld device will give you a better emulation experience. But if you want to try it out for yourself, follow the instructions here. Veund

Veund

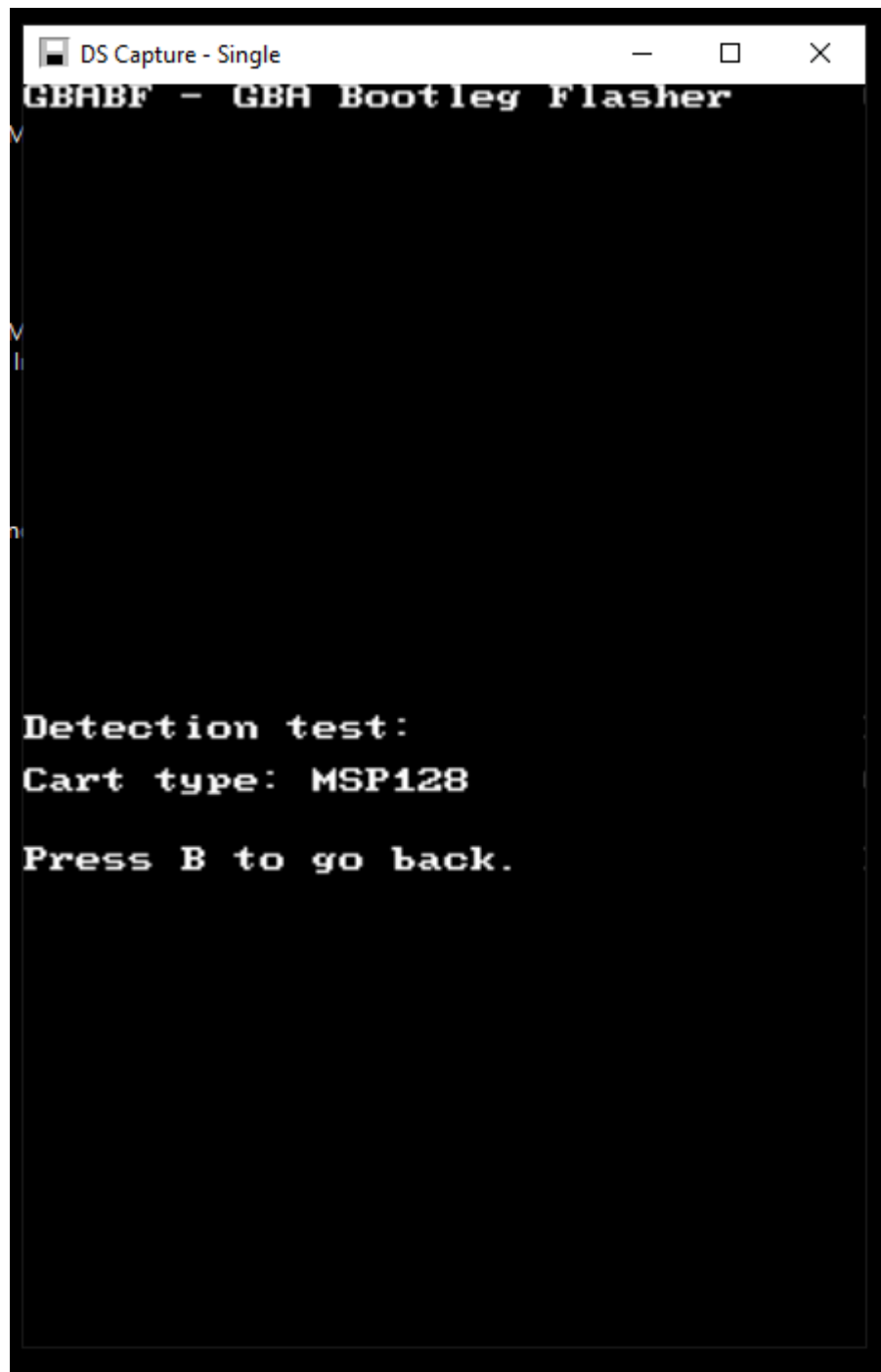
Game Boy Enthusiast, Hardware Fanatic, Mad Man. (he/him) ### Share this:

- [Click to share on Bluesky \(Opens in new window\) Bluesky](#)
- [Click to share on X \(Opens in new window\) X](#)
- [Click to share on Facebook \(Opens in new window\) Facebook](#)
- [Click to share on Mastodon \(Opens in new window\) Mastodon](#)
- [Click to share on LinkedIn \(Opens in new window\) LinkedIn](#)

- [Click to share on Reddit \(Opens in new window\)](#) [Reddit](#) [### Related](#)
-

Reflashing GBA Bootlegs with an NDS or NDS Lite

Recently I stumbled across a piece of homebrew called *GBABF*, it allows you to reflash GBA bootlegs with whatever ROM you like. In the past, we have written two articles on making GB Studio GBA formatted ROMs (one using *GB Senpai* and another using Goomba Color Emulator). What is great about GBABF is that an NDS or NDS Lite are fairly accessible to most people and don't cost an arm and a leg. Here is what you will need: * GBABF * NDS/NDS Lite * R4 or similar device * GBA Bootleg * Your GBS ROM formatted for GBA (see above links) An NDS or NDS Lite can be picked up pretty cheap but price may depend on your regional location. An R4 can be picked up for under \$10 on aliexpress and so can GBA Bootlegs. ## **Using GBABF** Simply copy GBABF and your GB Studio GBA formatted ROM to the SD card, insert it into your R4. Then place your R4 into slot 1 and your GBA Bootleg Cart into slot 2 of the NDS. When you boot GBABF, you are given a number of options. You want to start by selecting "*Detect Flash*". This is going to tell you if your bootleg is supported. From my testing of a small sample size of 5 bootlegs, 3 of them were detected.



GBABF testing the cart type of the Bootleg GBA Cartridge If your boot-

leg is detected you can then flash your ROM. Now choose “*Flash Rom*” . This will detect any GBA ROM on the flash cart so you don’ t even have to remember where you put it. Select your ROM and flash your bootleg.



Easy as that! GBABF is not the fastest, so if your ROM is on the bigger side it may take a little while, but when it's done you will have your new GBA bootleg to give to your friends, play on a GBA or even sell! GBABF opens up flashing carts to a wider variety of users on a budget. Sadly, its incapable of flashing GB bootlegs, as GB carts don't physically fit into the cart slot. It's on native hardware but it is still another option for the broader homebrew community. Veund

Veund

Game Boy Enthusiast, Hardware Fanatic, Mad Man. (he/him) ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### *Related*

Running My GB Studio Games on Xbox

The original Xbox has to be my favorite home console of all time. The Xbox started my journey into modding consoles many many years ago, and with the Game Boy being my favorite handheld, I had to work out how to get our GB Studio games working on the Xbox. This will require you to have a modified original Xbox that is outside the scope of this article, but both a hard-modded Xbox or a soft-modded Xbox will work. It is highly recommended to mod your Xbox in 2023 due to the high failure rate of various original components of the Xbox. The original Xbox has a massive homebrew scene. Even in 2023, there are still massive developments being made for the console. There are also a lot of different ways that you can set up an Xbox so this is less of a “how to get your games running” and more of a “how I got them running”, with that said let's get into it... ## **My Xbox** As I mentioned there are a lot of different setups for the box, so I thought I'd quickly go over my setup. I have a number of Xboxes and some setups are specifically designed for emulation such as XBMC-Emustation, but in most circumstances modded Xboxes come running UnleashedX (which is what I'll be using). My Xbox is also hard-modded, running cerbios 2.0.3R and a 2TB HDD upgraded from the stock 8GB hdd. This is also a PAL console modified to output NTSC video signal.

My modded Xbox's home screen ## **Getting It To Work** Now that's out of the way, here is what I did. Just about every modded Xbox dashboard has a built in FTP server that allows you to use an FTP client to connect to your Xbox over a wired connection. I personally use Filezilla, but any program will work. First, I needed to work out what emulator I was going to use. There are a couple of options but I decided on Xboyadvance because it can also play

Gameboy Advance titles. Most dashboards have an apps option. I initially copied FTP Xboyadvance into F:/apps but this caused Xboyadvance to get stuck on a black screen, but when I changed it to E:/apps, I didn't have any issues. Be careful as the E partition contains information for your Xbox such as save data and DLC, and the Xbox can get grumpy if it's full. Next, I copied my GB Studio game to F:/Gameboy/ (note that there is a file name limit on the original Xbox, so be sure to keep file names short and descriptive). I then launched Xboyadvance and navigated to my game folder on the F and launched my games. Xboyadvance had no issues reading games from F drive, so perhaps it was a bad transfer that needed the app to be placed on the E drive. The Xbox only has 10/100 mbps ethernet, not gigabit ethernet, which is what most people are used to, so this can take some time to transfer especially if you plan to transfer multiple games or files to the console at once. It's also quite possible that your Xbox doesn't have an F partition, this is normal because an Xbox doesn't usually have an F partition, this is only added if you have an upgraded HDD like I have, and in some circumstances you might even have a G partition. You also want to make sure you don't copy anything to X, Y or Z partitions, these are cache partitions and will be wiped eventually. ## **Final Thoughts** The Xbox is a great emulation machine capable of playing PS1/N64 games so there is no surprise that our games run really well on the hardware and it adds another way that people can enjoy our games. Veund

Veund

Game Boy Enthusiast, Hardware Fanatic, Mad Man. (he/him) ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### *Related*

The Goomba Color Emulator

What is Goomba Color and why is it important that GB Studio ROMS work in it? Goomba Color is a Game Boy/Game Boy Color (GB/GBC) emulator for the Game Boy Advance (GBA). That might sound like an odd statement as the GBA already plays GB/GBC games. But there are actually 3 consoles that are able to play GBA games but not GB/GBC ones: the Gameboy Micro, Nintendo DS and Nintendo DS Lite. Using it will allow you to run your games on these consoles, which is great if these are all you have access to (or just prefer to play on them). # How it works You essentially inject your GB Studio ROM into a Goomba Color emulator ROM, then place that onto a GBA flash cart, and it will run on one of the consoles mentioned above. There are 3 options for the Goomba Color emulator. Which option you use will depend on the cart you are going to flash. The official

Goomba Colour emulator saves to SRAM, so your cart will require a save battery if you wish to use saves. The second option is a fairly recent development for the Goomba Color emulator. BennVenn, whose work has been featured previously on GB Studio Central, picked up the Goomba Color emulator as a project and modified it to save to FLASH instead of SRAM. This only works on some carts which are available on his site, and there is also a link to the modified Goomba Color emulator there. <https://bennvenn.myshopify.com/collections/flash-carts/products/gba-flash-cart-16mbyte-128kbytes-sram> Shortly after BennVenn picked up Goomba, so did someone who goes by the name of Lesserkuma. Their version supports a tonne of different carts and more are being added as time goes on. This also has more on screen prompts than the BennVenn one. <https://github.com/lesserkuma/goombacolor> # Injecting Your ROM Into Goomba This is actually super easy to do, navigate to the Goomba's site and grab the latest Goomba Color files. It's Windows only as far as I'm aware, sorry Mac and Linux users. <https://www.dwedit.org/gba/goombacolor.php> There should be a folder with 3 files in it: **gbctrim.exe**, **goomba.gba** and **goombafront.exe**. If you are using BennVenn or Lesserkuma's version, replace the goomba.gba file in this folder with the modified one. Next, launch goombafront.exe and select the "goomba.gba" emulator file that's inside the Goomba folder, then select a destination and name for your output file. Under "add GB/GBC Games", navigate to your ROM location (in my case Z:\Goomba) and all the ROMs in that folder will appear in the box underneath. Highlight the ROM you wish to use and click Add. The game will appear as its name in the ROM's header, not the file name. You can remove any old ones from the list and also rename it by highlighting it in the list and right-clicking. When you click compile, it will create a Game Boy Advance ROM that you can transfer or flash to a Gameboy Advance cart for running on your console.

Veund

Figure 2: Veund

Veund

Game Boy Enthusiast, Hardware Fanatic, Mad Man. (he/him) ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### *Related*

VII. RPG Series Tutorials

- Creating A "Reward" Scene

- Let's Build an RPG | Quest 1-1: Battle System Player Turn
- Let's Build an RPG!
- Let's Build an RPG: Quest 2 - Rewards & Level Up
- Let's Build an RPG | Quest 1-2: Enemy Turn, Effects, and Stats
- Let's Build an RPG | Quest 1-3: Animations & HUD
- Quest 3: Back to Town Part 1 - The Inn

Let's Build an RPG: Quest 2 - Rewards & Level Up

The RPG Series

RPGs are one of gaming's most popular and longest-lived genres, and some of the most popular series in the world have had entries (or even got their start) on the Game Boy. And now, thanks to GB Studio, it's your turn to create an RPG of your own!

In this series, we will cover how to use GB Studio's built-in events to create a basic RPG system that you can expand on for your own game. All project files will be available to download at the end of the article, and inside you'll have all the examples and assets we use in each article (along with some additional comments). You can use these as a reference or even expand on them to create your own game. ## Quest 2: Rewards & Leveling Up With our battle system's core mechanics and flow outlined in Quest 1, we'll move on to fleshing out the resolution of a battle to prepare for the rest of the game. In today's quest, we'll cover how to provide the players with rewards for winning a battle and how to handle defeat. ### Adding a Game Over Screen To differentiate our battle results, instead of sending our player directly back to the Boot Screen, let's send our player to a Game Over screen instead. This will also begin to prepare our game for when we have more options (such as loading a saved game or quitting back to a main menu).

First, we'll add a new scene called "Game Over." Adding our game over scene. Back in our Battle Scene, we'll update our Monster Turn sequence to now send us to this scene when we are defeated, and we can spice up the sequence a bit by adding some music and a simple screen fade.

In GB Studio 4, dialogue events were improved to add more advanced features to control how your text displays. For our "You Died..." message, we'll set this to text box to close "Never (Non Modal)" – this will keep the text displayed while our script continues to run, allowing us to time the music with a wait event and not need to wait for the player to press A. As our game over music plays, the screen will fade out, and take us to our new scene. Our game over script. We'll need to use the new **[Close Non-Modal Dialogue]** event before we change our scene to make sure the text box we kept open doesn't reappear on our next scene, too.

For now, we'll keep our Game Over scene very simple. We'll give the player a choice to continue using the **[Display Multiple Choice]** event, but for now

we’ ll have both answers return to the Boot Scene. We’ ll come back to this later on in the series! Prepping our game over sequence for a future article. Now that we’ ve added that, don’ t forget to update any other scripts that check for the player’ s defeat (such as in our Poison damage script!). ### Creating Our Victory Sequence Next, we’ ll update what happens when we defeat the monster. Right now, we only need to update the sequence in our “Fight” trigger.

Currently, once the enemy has reached 0 HP, we just send the player back to Boot Scene – kind of anti-climactic, isn’ t it? Just like our Game Over sequence, let’ s spice that up by adding some music. We’ ll also want to use [**Hide Actor**] on our enemy sprite, since we’ re no longer going to immediately leave the scene. Checking for victory. Instead of sending the player to a new screen for our Victory sequence, let’ s add a new trigger (named “Victory!”) to handle this script as part of our trigger based menu and send the player there. Creating our victory trigger. ### Rewarding Victory With our victory trigger created, it’ s time to hand out some rewards! In most RPGs, defeating monsters will reward gold, items, and experience points (XP). We’ ll set up our battle system to reward all three, but you can tailor the types of rewards to different types of battles if you’ d like. #### Gold Let’ s start with giving the player some Gold for defeating our monster. First, create an [**Event Group**] to keep your script organized. In order to make sure our player doesn’ t earn more gold than they can carry, we’ ll first check if they already have max gold. For this, we’ ll assume we’ re limiting the player to 9,999 gold, but you could also use a variable here if you wanted to allow the max gold to change over time (but remember, a variable can only go as high as 32,767).

If our player doesn’ t already have 9,999 gold, we’ ll use a [**Math Function**] to reward a random amount of Gold. Let’ s set one of our trigger’ s local variables (renamed to \$Gold Rewarded in our examples) to this amount. Then, we’ ll need to use a [**Math Expression**] to actually update our player’ s Gold.

Create a global variable (in our project, this was variable 13) named “Gold.” With our Math expression, we’ ll see if the amount rewarded to make sure our gold doesn’ t go above 999 using the “ min(“ function. This will reward the lower of the two options – either our “Gold + Gold Rewarded” , or 9,999. Add a few dialogue events to tell the player their reward (or lack of reward), and we’ re all set! Our reward script for gold. #### Items Managing an inventory can be complicated, and there are many different ways you can create an item and inventory system. For our game, we will be giving our player 10 inventory slots to hold items, and each will take up a whole slot in our bag.

First, create 10 global variables, named Item 1, Item 2, and so forth. These are what we’ ll use to see if our player can hold an item. We’ ll also need to determine what our variables represent. For example, while Item 1 = 0, we know this inventory slot is “empty,” but once it has a value it will represent a different item.

As we introduce more items and shops into our game, we can determine what

these values represent. For now, we'll assume that a value of "1" is equal to a "Potion."

In our Item event group, we'll first want to check if we have an open inventory slot to give the player an item. To do this, we'll need to stack a series of [If Variable False] events, checking if our Item variables are empty.

If Item 1 is false ("0"), we can reward the player with the item and store it in that variable, but if Item 1 was true (any non-zero value), we'd use the "Else" function of our event to check our next variable. Stacking events to create our item checks. We'll repeat this for all 10 Item variables. If all our Item variables are true, in our final "Else," we can add a dialogue box that tells the player they didn't receive an item because their bags are full.

To actually set the variable, we'll use a **[Set Variable to Value]** event at the start of our script. By setting a local variable (renamed Item Rewarded in our example) to a value, we know that this monster will reward that specific item (remember, a "1" = a potion), and by using a local variable, this makes it easier for us to change this in the future – if we copied this event for another monster, we could change the items that that monster gives out, or we could update this in the future to give a random item. Setting our item value. In each of our Item checks, we can then use another **[Set Variable to Value]** event to set the Item global variable to the Item Rewarded; this just simplifies the repetition – we don't need to go and change every event to a new value if we wanted to change what item is rewarded; instead we'd just need to change the first event. ##### Experience points Finally, let's give the player some Experience Points (XP). Like most RPGs, defeating monsters in our game will reward XP, and after a certain amount of XP is gained, our player will increase their level and grow stronger.

First, we'll create three more global variables: Hero XP, Hero Level, and Monster Level. For our game, we'll use a very simple formula to determine how much XP a monster is worth to our player: if a monster is at a higher level than the player, we'll give out double XP, but if our player is at a higher level than a monster, we'll give out half the XP.

In our Battle Scene On Init script, we'll add a new event in our "Set Monster Stats" event group to set our Monster's level to 2. Setting our monster's level. In our Boot scene, we'll also want to set our player's Level to 1 and their XP to 0. Setting our starting XP and level. Back in our Victory trigger, we'll add our XP formula. First, we'll determine how much XP our Turnip is worth – let's go with 10 XP to start. Using a **[Set Variable to Value]** event, we'll set a local variable (renamed XP Rewarded) to 10.

To set up our formula, we'll use **[Compare variable to variable]** events. First, we'll check if the monster is higher level than our player, and reward double the XP Rewarded variable if that's true. In the Else, if the Monster Level is lower than the Hero Level, we'll use another Math Expression to half the XP Rewarded variable. And if the Hero Level and Monster Level are the same,

nothing will change. Checking our level vs the monster's to determine a reward. After that, add your XP Rewarded to your Hero XP using a Math Expression. Our XP formula. Finally, we can add a dialogue event to show how much XP the player earned, and with that, we have a simple XP reward script. Displaying a reward dialogue. Nice and easy! But what now that we've earned that XP, what do we do with it? ### Leveling Up To check if our player has earned enough XP to level up, we'll use a simple formula to increase the amount of XP needed every level. This way, our player progressively needs to earn more XP every level.

Our formula will be $\$Hero\ Level * 100 * 1.25$, which gives us the following progression for experience points: | | | ***

Let's Build an RPG | Quest 1-2: Enemy Turn, Effects, and Stats

The RPG Series

RPGs are one of gaming's most popular and longest lived genres, and some of the most popular series in the world have had entries (or even got their start) on the GameBoy. And now, thanks to GB Studio, it's your turn to create an RPG of your own!

In this series, we will cover how to use GB Studio's built-in events to create a basic RPG system that you can expand on for your own game. All project files will be available to download at the end of the article, and inside you'll have all the examples and assets we use in each article (along with some additional comments). You can use these as a reference or even expand on them to create your own game. ## Quest 1, Part 2: Enemy, Effects, and Stats Now that we have the basic turn sequence established, let's make this start functioning like a real battle. An RPG wouldn't be much of a game if the monsters just stood there looking menacing, now would it?

In today's article, we'll add a monster to our battle scene, give it two types of attacks, and also start tracking some basic stats for the character & the monster like HP, Attack Power, and more.

Let's go! ## Character Creation In order to begin tracking stats for our game, we're going to need to utilize variables. If you're new to GB Studio, you may want to check out our Basics series on Variables, as they're the backbone of almost every kind of game.

To cover the basics of our Battle, we'll want to have a few basic stats, but we'll be adding more down the line. You can also customize these as you get a better idea of how you want your game's progression and various formulas to work, but for this series' purposes, we'll assume you're following this basic setup.

Let's start setting up our main character. Head over to the Boot Scene we created in Quest 1. In the ON INIT tab of the scene, create an [Event Group] to

contain our player stats – this will help with organization, but has no function on its own.

Since our ON INIT tab already has some events, you’ ll notice that the “Add Event” button is beneath the current scripts. You’ ll want to make sure this event is placed before the scene change – any events placed after a [**Change Scene**] event will not run! You can do this by dragging the event around to change the order, or by clicking the down arrow in the corner of an existing event and selecting “Insert Event Before” . Once you’ ve created your event group, add a [Set Variable to Value] event inside:

I’ ve renamed my event group to help with organization. The [**Set Variable to Value**] event defaults to a Local variable; since our player stats will be used in more than just this script, we’ ll want to set these to a Global variable. You can use the dropdown box to select an unused Global Variable (in this project, it’ s Variable 0). Then, let’ s rename it so that we can remember what the variable controls when we use it later on.

We’ ll need to create a few stats, so create a [**Set Variable to Value**] event for each one, using a new Global variable for each stat. For this article, we’ ll create:

HERO MAX HP

Our Max HP will be used to determine our total HP, and will be useful for keeping track of HP when we need to do things like heal. Let’ s set this to 20 to start.

HERO CURRENT HP

The Current HP will be what we actually modify when we take damage. Since we’ re starting out at full, we’ ll want to set this to 20 (our Max HP).

HERO ATK

Our ATK stat, or “Attack” , will be used to determine if our fight actions are successful. Let’ s set this to 5.

HERO DEF

Our DEF, or “Defense” , will be used to check if an enemy’ s Attacks are successful. Let’ s set this to 3.

HERO SPEED

Our SPEED stat will be used to check success when we try to flee from battle. Let’ s set this to 4.

HERO STATUS

Finally, our STATUS will be used to track what kind of special conditions we’ re affected by, like being Poisoned. Since we’ re starting out healthy, we’ ll set this to 0.

Once your stats are set up, your event group should look something like this: But wait! In this screenshot, our Hero Current HP stat isn’ t set to a value – it’ s set to another variable! Since our game is just beginning, our Hero’ s

“Current” HP should be equal to the “Max” , so we can use a function of the **[Set Variable to Value]** event to do this. When we set our Hero Current HP, click the # symbol next to where we enter a variable, and from that drop down, we can select a Variable instead of a Value. Select the “Hero Max HP” variable, and now when our game begins, our current HP will start at the Max!

This will be helpful in the future! With our Hero’ s stats confirmed, let’ s move on to our Monster. ## A Monster Appears Unlike our Hero, a Monster’ s stats won’ t need to be set up during the start of the game. These will be set when a battle begins, so let’ s head back to our Battle Scene.

In the ON INIT tab of the Battle Scene, we’ ll create an event group for setting up our monster just like we did for the Hero. In that event group, we’ ll use **[Set Variable to Value]** events to assign values to the stats in the same way. Note that even though these variables will only usually be used in a Battle scene, they will still need to be GLOBAL variables, as we’ ll want to use them in other scripts. Luckily, we can re-use these in other battle scenes, so we won’ t need to worry about different monsters duplicating variables. ### Monster Actions Now that our monster has stats, let’ s make it do something on its turn. To keep things interesting, we’ ll give our monster a Normal Attack, but there will be a small chance for it to sometimes do a Special Attack.

First, let’ s head to our Monster Turn trigger. You can delete the placeholder dialogue from earlier.

We’ ll start by using a **[Math Functions]** event to generate a random number between 1 and 8. I used Local 0, and renamed it to “Monster Behavior” .

Next, use a **[Compare Variable to Value]** event to see if the value of that variable is greater than 2. If it is, we’ ll have the monster do a Normal Attack. But if it rolls low, we’ ll have it do a Special Attack. ### Normal Attacks In our previous article, our combat formulas were simple dice rolls. Now that we have assigned some basic stats, let’ s start fleshing those out, beginning with our monster’ s Normal Attack. To start using our variables in our formulas, we’ ll want to use Math Expressions.

Combat formulas can come in all shapes and sizes, and working out how your stats your will impact your game is part of the fun of creating an RPG. For our tutorial, we’ ll be using a simple formula influenced by common tabletop RPG mechanics.

To determine if an attack hits, we’ ll have the attacker make an attack roll, which will be another **[Math Functions]** event. For this, we’ ll set a variable to a random value between 1 and 20. But instead of having a static target value to compare our result against to determine success or failure, let’ s start factoring in our stats.

Create a **[Compare Variable to Variable]** event, and using the drop down fields next to their value entry, select “Expression” . This will allow you to write a Math Expression in place of entering a static number.

To get our Monster's attack power, we'll combine the Attack Roll variable, our Monster Speed variable, and our Monster ATK variable. Then, we'll compare that against the Hero's DEF, plus their Hero Speed, and a flat defense value of 10.

Note: To use a variable in an expression, begin typing a \$ followed by a variable name or number. This will let you select a Local, Global, or Temp variable for your formula! With this combat formula, our Hero has a defense value of 17. Our monster's ATK and Speed are equal to 6, so we'll need to roll an 11 or higher to hit the Hero, meaning we'll hit just under 50% of the time.

If we do hit, we can deal damage to the Hero. Our damage formula is straightforward; we'll use another **[Math Expression]** to set the Hero Current HP to "Hero Current HP - Monster Attack". We can use **[Display Dialogue]** and **[Camera Shake]** to add a little flavor to our attack sequence, too. By displaying the Monster Attack variable in dialogue, we can also show how much damage the player took, and if we miss, we'll show a message using **[Display Dialogue]**.

A basic attack formula and outcomes. ### Special Attacks For our special attack, let's have our monster try and give the hero a status condition! We can use many of the same functions we just built, so following the same method as our Normal Attack, create an attack roll and comparison event. Instead of having our hero take damage on a Hit, we'll have it change their Status variable, but only if they don't already have a status condition.

To do this, create an **[If Variable is False]** event, using the Hero Status variable. Then, in the event, add a **[Set Variable to Value]** event to change the Hero Status variable to 1. This will assign a status, which we'll decide is the "Poison" status. Since we can't rename a value, we'll have to write this down and for future reference. A comment event can be helpful to record what our values represent, or a separate text file in our project folder can keep track. If the player already has a status condition, we can add a **[Display Dialogue]** event or other way to show that nothing happened, just as if we missed. ### Ending the Turn With our Monster's attack resolved, we'll need to see if the player was defeated or if we'll continue into a new turn.

Using a **[Compare Variable to Value]** event, create a check to see if the Hero Current HP is less than or equal to 0. If it is, we'll send the player back to the Boot Scene (eventually, we'll have a game over screen later on), but if they still have HP remaining, we'll start a new turn by sending the player actor back to our Player Turn trigger. ### Status Effects Now that our player can be poisoned, we'll need to make that take effect. We'll have poison damage happen at the start of the player's turn, so head over to the Player Turn trigger.

Before our Menu event, add an **[If Variable is True]** event, selecting our Hero Status variable. Inside, we'll place a **[Switch]** - right now it will only have 1 option, but we'll add more later on. Let's make poison damage deal a random amount of damage. You know the drill - add a **[Math Function]**, and

have the range be 1 to 3. Since we don't need to keep track of the poison damage outside of this event, we'll use a Temp Variable to track our poison damage. Then, just like in our Monster's normal attack, we can add a **[Display Dialogue]** referencing \$Temp0 to show how much damage we took, and a **[Math Expression]** that sets Current HP to Current HP - Temp 0 to actually deal the damage. Using another **[If Variable Compare to Value]** event, check if the Hero's Current HP is 0 to see if we died, and if we didn't, we'll continue the turn as normal. And there you go, our status condition checks are all set! ### Wait, aren't we forgetting something? Hold on – if we're checking for poison damage at the start of our player turn, every time we enter that trigger, even if we cancel a menu choice, we'll be checking to see if we're poisoned. We've had our first bug, and that's no good!

Proper sequencing and controlling scripts is hard to get right, especially as our games get more complex. Luckily, we can fix this with a variable!

Using another **[If Variable is False]** event, let's make a check for a new global variable called "Status Check". If this is false, we'll run the script to check if we have a status condition, and when a status check happens we'll set this variable to True. This way, if we go back and forth in a menu during the player's turn, we'll only ever run this once! We'll set the status check variable to false at the end of the Monster's turn so that we can check it again on the next player turn. You'll also want to add this into our Boot Scene so that whenever we begin a battle, Status Check is set to false. ## Pop Quiz! Using what we learned with the Monster turn, we can go back to our player's Attack and Flee actions and update those scripts to use similar attack, damage, and other formulas. For example, to run from a battle, we could try a different calculation like a "roll off" against the monster to compare who is faster.

Try and put your new knowledge to the test and update these yourself; the project file has the updated sequences, so you can check your work against the download!

You can download this tutorial's project files below, or visit the project on Itch.io! Download ## Up Next: Wait a second...who's that mean looking guy in our Battle Scene? In our next article, we'll start to add some more sprites and effects to our scene, and dip our toes into GBVM to display our HP in a HUD!

This article is part of a series: Let's Build an RPG! Dan Enders

Dan Enders

Friendly Neighborhood Game Designer, Podcaster, and (ex)Librarian (he/him)
GB Studio Games | Tabletop Games ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon

- Click to share on LinkedIn (Opens in new window) [LinkedIn](#)
- Click to share on Reddit (Opens in new window) [Reddit](#) *### Related*

Let' s Build an RPG | Quest 1-3: Animations & HUD

The RPG Series

*RPGs are one of gaming' s most popular and longest-lived genres, and some of the most popular series in the world have had entries (or even got their start) on the GameBoy. And now, thanks to GB Studio, it' s your turn to create an RPG of your own!

In this series, we will cover how to use GB Studio' s built-in events to create a basic RPG system that you can expand on for your own game. All project files will be available to download at the end of the article, and inside you' ll have all the examples and assets we use in each article (along with some additional comments). You can use these as a reference or even expand on them to create your own game.* *### Quest 1, Part 3: Animations & HUD* Now that our combat system has the core features in place, let' s take a break from triggers and menus, and start making our battle screen a little more interesting! Visuals are more than just for looks, of course; we' ll need to communicate key information to the player (like how much HP they have remaining) so that they can make the right decisions during a battle.

In today' s article, we' ll add a few visual effects to our game, a monster sprite, and dip our toes into some basic tile swapping to display our player' s HP and MP *### Enemy Sprites* In our last article, our enemy made their debut: we' ll be duking it out against GB Studio' s classic enemy, the Turnip! Coming in 32 x 32 pixels and a healthy 6 unique tiles, the Turnip will make a formidable foe, but still leave us plenty of room to work with when considering our scene limits. A screenshot of the battle menu, showing off the Turnip enemy sprite. If you' re using our project files, you' ll find the Turnip located in the Assets/Sprites/Monsters folder, and it' s already been loaded into the sprite editor.

We' ve added this villainous vegetable into our Battle Scene as an Actor, and we can create some simple animations to give our attack actions a little more flair. *### Simple Actor Animations* Head over to the Fight trigger; in our "Hit!" sequence, we' ll use the **[Hide Actor]**, **[Show Actor]**, and **[Wait]** events to add a simple "flicker" sequence to confirm the hit. For an alternate method, or if you have a sprite that shows a damage animation, you could use the Sprite Editor to add an animation state to your enemy and play it here instead! A screenshot of the animation script. ***Note:** We' ve put the "Animation" event group in place of our placeholder comment from earlier articles!*

Next, let' s make the Turnip look like it dodges an attack when we miss. In our

“Miss!” sequence, we’ ll add an Animation sequence using the [Actor: Move Relative] event to move our Turnip actor 2 tiles to the right, and then 2 tiles back to its original position. An image showing how to use move events to animate our sprite. Let’ s see it in action: Showing the animation in action! ### Combat Effects Our battle sequence is already looking better, but let’ s take it even further. In our Assets/Sprites/Effects folder, you’ ll see we have a spritesheet called “slash” . It’ s already been loaded into the Sprite Editor, so let’ s use it to show that our attack occurred!

In our Battle Scene, add a new Actor called “Slash” to scene, placing it overlapped with our Turnip actor. A screenshot of our battle scene in GB Studio.

We’ ll want to make sure our effect only plays when we want it to, so we can use [**Hide Actor**] on its Init tab to always hide this at the start of our battle. Showing our Slash actor’ s scripts. Then head back to our Fight trigger. We’ ll place another animation event group at the start of the sequence, just after our dialogue event. In that event group, we’ ll add the following events:

First, we’ ll use [**Show Actor**] to make the Slash actor visible.

Then, we’ ll use [**Actor: Set Animation Frame**] to set the animation frame to 0. This will make sure the animation always plays from the beginning!

After that, we’ ll add a short [**Wait**] event. We set this to 9 frames, because that’ s how many frames are in our Slash animation (which was created in the Sprite Editor).

Finally, [**Hide Actor**] will make the Slash actor invisible again. Setting the Slash animation frames. With that sequence, we should end up with something like this: Showing off the slash animation. Perfect! Now our player has plenty of visual feedback for when we’ ve performed an attack. But what about when we get hit in return? ### Changing Background Tiles In order to update our HP numbers, and eventually our MP, we’ ll need to change our background, and we can do this by manipulating its tiles. A background is made up of a series of 8×8 tiles, and more recent versions of GB Studio have given us even more ways to interact with backgrounds in real time. In 3.0, we gained access to GBVM, which enabled players to start making HUDS and animating backgrounds relatively easily. But with the release of 4.0, we now have access to the [**Replace Tile at Position**] event, making this even easier!

To Replace tiles, we’ ll need to create a tilesets folder in our assets folder. A tileset is a lot like a background – it’ s a set of 8×8 tiles that we’ ll pull from to insert into our current background. We’ ve already created this folder in the project files for you, and there are 2 tilesets in at this point: *numbers.png* and *blank.png*.

Let’ s begin by updating our HUD when our battle begins. Head over to the On Init area of the Battle Scene. After we set up our monster, let’ s create an [**Event Group**] to display our HP. Starting the script for our HUD. In this event group, we’ ll be creating a series of [**Compare Variable to Value**] events

to control our “ones,” “tens,” and “hundreds” columns of our HP display. We’ll start with the “hundreds” column. ### The Basics of Replacing Tiles First, create the **[Compare Variable to Value]** event, and set the condition to check if our Hero Current HP variable is greater than 99. If it’s not, we don’t need to show a number here, so in the Else area, let’s use the **[Replace Tiles at Position]** event to hide the hundreds column. Setting up our script to hide our hundreds column tile. **[Replace Tiles at Position]** has a few parameters we need to enter. First, we’ll need to enter the X and Y coordinates of the tile we want to replace. The replace tile event. (***Note:** Replacing a tile will replace all instances of that tile in a scene, so always make sure you’re using unique tiles in your background for elements that you’ll be swapping!*) Highlighting our hundreds column tile.

Selecting the “hundreds” column Once you’ve set the coordinates, you’ll select a tileset from your `/assets/tilesets` folder. Then, if your tileset has more than 1 tile in it, you can select the tile in your tileset to use.

For this event, we’ll use the `blank.png` tileset, and since this is a single tile tileset, we can leave the tile number as 0.

Since our Current HP is not more than 99, a quick check when running the game should show this: A screenshot showing how the hundreds column is hidden in game!

We’ve hidden the 0 from the hundreds column! Now that we’ve got the fundamentals down, it’s time for some math! ### Using Math to Display our HP Since our HP is split into 3 columns, but we’re only using a single variable to store our HP value, we’ll need to parse out the individual components of our number to get our hundreds, tens, and ones values. We’ll also want to start working with Temp Variables so that we don’t accidentally edit our hero’s Current HP variable!

***Pro Tip:** Temp Variables cannot be used in custom events; if you’re working on your own and utilizing custom events, you can create global variables to use in their place!*

Let’s head back to the Compare Event – if our hero’s HP actually was above 99, we’ll need to display that number. Using a **[Set Variable to Value]** event, we’ll set **Temp 0** to our Current HP variable. This will let us manipulate the value without actually changing our HP variable for other uses.

Once we’ve set that, use a Math Function to divide that number 100. This will get us the single digit we need to show our hundreds column! A screenshot showing how we use Temp variables in our math events. If we change our hero’s HP back in the boot scene to a number like 123 for example, here’s what would display if we ran the game: A screenshot showing how the hundreds column is updated with a new tile.

Displaying the hundreds column With the hundreds column solved, we’ll need to move on to our tens and ones.

Create another **[Compare Variable to Value]** event below our first. Here, we’ ll want to see if the HP is above 9 – if we’ re not, we can use the **[Replace Tiles at Position]** event to hide the “tens” column like we did earlier – select the coordinates and use *blank.png* to hide that number. A screenshot of the replace tile event. If we are above 9 though, we’ ll need to get the right value. Just like before, we’ ll assign the Hero Current HP value to a Temp variable, and then we’ ll use a Math Function to find our value. To do this, we’ ll need to use the Modulus math function. Using the modulus math function. To break this down, Modulo gives us the remainder of a number after dividing by that amount. Since our HP in this scenario is 100 or above, we’ ll enter 100 so that we divide by that amount, but are given the value left over. If our HP was 123, for example, we’ d be given 23.

Since we’ re looking for the tens column though, we’ ll need to divide that remainder again by 10 so that we’ re left with a single digit. Dividing our variable with the math function. Then, we can replace our tile with the **[Replace Tile at Position]** event, just like we did for the hundreds column. A screenshot of the full script to replace our tens column tile.

The full “tens” script Continuing our example with an HP value of 123, your HP should now display a 1 in the hundreds column and a 2 in the tens column, like so: A screenshot showing off the replaced tens column tile. With that new knowledge, let’ s finish off the “ones” column. We’ ll need to run this operation twice – first, you’ ll need to use the combo of Modulo to get the tens column like we just did, and then you’ ll need to use another Modulo to get the ones column digit! The ones column script. Note that since our HP can never be below zero, in our “else” section for our ones column, we’ ll use *numbers.png* as the tileset instead of *blank.png*.

Now, when running the game, you’ ll see a full HP value! A screenshot of all our number tiles being updated! If we copy this event group into our enemy’ s attack sequence (or wherever we update HP, like our Poison Status script), we’ ll see these numbers update in real time as we take damage! Showing off all the tiles being updated. That’ s a lot of damage! But the HUD looks great, and with the animations added, we’ re looking even more like a real game! ## Moving On This wraps up the basic actions for our battle system, but there’ s still plenty more to come. In our next article, we’ ll start tracking XP and other battle rewards to start preparing us for our first trip to town! Don’ t forget to download the project files to see more details, and visit the [itch.io](#) page for the project to leave questions and comments. You’ ll also find some more information on this series’ collaborators below! Download ## Special Thanks The amazing animations and fearsome enemy sprites that we’ ll be using for the remainder of the series were provided by Eric Mack. As the series continues, we’ ll add more awesome animations and enemies, expanding on the style seen in GB Studio’ s Sample Game. If you’ ve been downloading the project files with each article, you’ ll notice that I’ ve slipped some music and sounds into our scenes. This tutorial won’ t be covering how to make your own music, but

don't worry! GB Studio Central has you covered with another great tutorial series! There's also great articles covering sound effects as you expand and customize your own game!

Our game's music is provided courtesy of Beatscribe's Free Music pack, and our attacks and damage actions are selections from their free Sound pack.

This article is part of a series: Let's Build an RPG! Dan Enders

Dan Enders

Friendly Neighborhood Game Designer, Podcaster, and (ex)Librarian (he/him)
GB Studio Games | Tabletop Games ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
 - Click to share on X (Opens in new window) X
 - Click to share on Facebook (Opens in new window) Facebook
 - Click to share on Mastodon (Opens in new window) Mastodon
 - Click to share on LinkedIn (Opens in new window) LinkedIn
 - Click to share on Reddit (Opens in new window) Reddit ### *Related*
-

Quest 3: Back to Town Part 1 - The Inn

The RPG Series

RPGs are one of gaming's most popular and longest lived genres, and some of the most popular series in the world have had entries (or even got their start) on the Game Boy. And now, thanks to GB Studio, it's your turn to create an RPG of your own!

In this series, we will cover how to use GB Studio's built-in events to create a basic RPG system that you can expand on for your own game. All project files will be available to download at the end of the article, and inside you'll have all the examples and assets we use in each article (along with some additional comments). You can use these as a reference or even expand on them to create your own game. ## Quest 3: Back to Town Part 1 – The Inn Now that our battle system has a way to hand out rewards, we'll need to actually give the player the opportunity to do something with all that money! In Quest 3, we'll create a town scene for our player to explore and provide various services. In our first part, we'll begin with a classic RPG staple – the Inn. ### Creating a Town We'll start off by creating a small town for our player to explore. If you're following along with our project assets, you'll see a few new scenes added to our project: a Town, an Inn, a Shop, and a mysterious Cave. We've borrowed these maps from the GB Studio Sample Game (with a few modifications – thanks Chris!).

The classic Sample Town In your typical RPG, Towns offer a player opportunities to rest after exploring, acquire new items, take on quests, and learn more about

the world or story of a game. Don't worry about fleshing out your world just yet – we've populated ours with a few NPCs and events for fun, but for now we'll be focused on the basic functions of a town.

Let's start with the Inn. ### Rest and Relaxation Inns are a staple of RPGs, and usually provide two key functions for a game. The first is to give the player a place to restore their HP and MP without expending items (usually for a modest fee), and the second is to provide a place to save your game. For our game, we're going to have the Inn provide both features!

Okay, so it's not the most inviting looking inn... In our Inn scene, we've placed a trigger near our Innkeeper NPC. Since he's behind a desk, and we've set that as a blocked area, we'll need a way to interact with that Actor. To do this, we'll use a different actor to run our scripts, and the Innkeeper sprite will be purely cosmetic. #### *Designer's Note: If you need to conserve actors in a scene, you could also accomplish this by creating a sprite that has its collision area below the character, allowing you to interact with it further away (a trick we'll use in our next article). You could also utilize triggers and the **[Attach Script to Button]** event, but since this is a simple scene, we can afford to commit an extra actor in this instance.* First, in the On Init area of our Innkeeper Object, we'll create a Local variable to use as the cost of the room. We'll use a variable to do this so that we can easily reference the value in dialogue and math events, as well as reuse the scripts for other Inns and easily adjust the price! After that, we'll use **[Hide Actor]** so that we don't actually see the arrow sprite. In the On Interact section, we'll add some dialogue to act like the Innkeeper. After that, we can get into our actual Inn event script.

Using a **[Display Multiple Choice]** event, we'll give the player the choice to stay at the Inn or not. If they do choose to stay, we'll need to make sure they can afford the room. With a **[Compare Variable to Variable]** event, we can check if the player's **\$Gold** (Global variable) is greater than or equal to the **\$Inn Cost** (Local Variable).

If the player has enough gold, we'll use an **[Evaluate Math Expression]** event to deduct the cost from our gold (and using the **max()** function we'll make sure we don't go below 0). Then, we can run the Inn's functions.

For a little flavor, we'll **[Play Music Track]** and **[Fade Screen Out]**. Then, while the screen is black, we'll **[Set Player Position]** to make it look like we used the Inn, and then we can **[Fade Screen In]** once our player has been moved to complete our sequence.

We'll also use this opportunity to restore our player's HP and MP. Using the **[Set Variable to Value]**, we'll set **\$Hero Current HP** to the **\$Hero Max HP** variable. We can do the same for MP (*Note: If you're using the project files, we've created those variables for this Quest, but haven't implemented them in our Battle system just yet*).

Once we've added all the effects of sleeping at the Inn, we can use **[Game**

Data Save] to save the game. Now, whenever a player starts the game, we can start from this Inn instead of starting fresh.

Our new Inn Sequence script **### Continuing the Quest** Since we’ ve now given the player the ability to save the game, let’ s take advantage of that and add in a Continue system.

Head over to our Boot Scene. Since this is where our project begins, we’ ll repurpose this scene as our Title Screen.

Now with a fancy new background! In our On Init section, we’ ll re-organize our boot script and add in the option of continuing our game.

First, we’ ll still want to **[Hide Player]**, and to make sure we can keep our Random Seed generating properly, we’ ll also want to **[Pause Script Until Input Pressed]** after our **[Fade Screen In]** event. Now, once the player presses Start, we’ ll use a **[If Game Data Saved]** event to check if there is any data saved in Save Slot 1. If Save Data is found, we’ ll want to give the player the chance to either continue their game or start a new one. Using a **[Display Multiple Choice]** event, we’ ll set a choice variable to be true if they continue, or false if they want to start a new game.

Then, using an **[If Variable True]** event, we’ ll **[Load Game Data]** if they chose to continue.

If no data is found, or if the player chose to start a new Game, our choice variable will be false. In that case, we’ ll have the game start a New Game automatically, and we’ ll run all the events we ran in previous articles as part of our original ‘boot scene’ – setting up our starting stats, gold, etc. Then, we’ ll change the scene to the start of our game, which for now we’ ll change to being a location in town. **#### Designer’ s Note:** *We’ ve made a few changes to our text boxes throughout the game just to make things a little easier on the eyes. Take advantage of GB STUDIO 4’ s **Behavior Presets** feature in dialogue events to customize your settings and differentiate between things like Battle messages and NPC dialogue.* **## Coming Up:** Our heroes are now well rested and ready to head back into town for a shopping spree! In Part 2, we’ ll create our Shop scene and learn how to buy items.

You can download the project files here, and don’ t forget to visit the project page on itch.io to ask questions or provide feedback. See you next time. RPG Quest 3Download

This article is part of a series: Let’ s Build an RPG! Dan Enders

Dan Enders

Friendly Neighborhood Game Designer, Podcaster, and (ex)Librarian (he/him)
GB Studio Games | Tabletop Games **### Share this:**

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X

- Click to share on Facebook (Opens in new window) Facebook
 - Click to share on Mastodon (Opens in new window) Mastodon
 - Click to share on LinkedIn (Opens in new window) LinkedIn
 - Click to share on Reddit (Opens in new window) Reddit ### *Related*
-

VIII. Game Genres: Puzzle

- Let's Program a Puzzle Game! - Part 1
- Let's Program a Puzzle Game! - Part 2
- Let's Program a Puzzle Game! - Part 3

Let's Program a Puzzle Game! - Part 2

Setting Up The Electric Slide

With all our movement set up, it's time to transition the puzzle into its proper form. The artwork we currently have set up is great, but we need to make those tiles slide and move around. Considering the limitations of GB Studio, we quickly encounter two key challenges: a restriction of only 10 actors in 160×144-sized scenes and graphical issues when actors overlap during movements (in DMG or non-color mode).

For that reason alone, we need to prepare our graphics to use tile-swapping for our puzzle. Starting with the background, we will need unique tiles for each one we are swapping, otherwise if we swap one, we swap any other identical tiles within the scene. So instead of our previously complete-looking artwork for the puzzle, we end up with something that looks like this: An image of the scene background with unique tiles to set up our puzzle.

Dr. Frankenstein's monster has nothing on this mess of numbers. Yes it's a jumbled mess, but it guarantees unique tiles, and that's all that matters. Another option is to put pixel dots on each tile, starting from the top left, moving to the right down to the next row, and so on. This would leave you with 64 unique tiles (8 columns by 8 rows), which is exactly what we need for this puzzle. You may wish to personalize the art to define your puzzle intentions, of course, this is just how I would handle the problem.

Now that our background is set up, we will need to transfer all our tiles for the puzzle to their new background. I always try to place the “blank” tile in the top left of the layout of this background, and then follow it in the easiest pattern. Since we have 15 tiles to move around, we'll place those next to it and follow to the next row. The tilemap used for the puzzle tile swapping.

Our Tilemap background for the Tiles we use in our puzzle. Those of you who have used tile-swapping before, perhaps with the older ways, will instantly cry foul with this layout, as you can immediately tell there are duplicate tiles. While that might be true, there's a way to avoid complications. By setting the new

scene type to [Logo], we make each tile unique, regardless of content. While it seems like we could use the same trick in the main puzzle scene, we need actors to handle our tile-sliding animations and some contextual text, and Logo scenes don't support actors.

Another important distinction is even though we are tile-swapping, we will use the Swap Tiles Var plugin to handle the process. Instead of putting our Tile-swapping background in our tileset folder inside the assets folder within our project, we should put them in the backgrounds folder instead.

We also need a sprite to handle the tiles we move using a separate actor. You'll set up the "slide_numbers" sprite as in the following screenshot, noting the Animation Type and Canvas Size. For Animation Type, we want to set it to 'Fixed Direction,' and our Canvas Size is set to 16×16. Another thing you will need to set up is the frames, leaving the first one blank, and then adding new frames and moving each of the 15 16×16 squares into the frame. A screenshot of the GB Studio sprite editor.

Animation Settings for the "slide_numbers" sprite You will need to add this new actor to the scene, and you can place it wherever you want, but I chose to put it in the bottom-right portion. Set it to the sprite we just set up in the previous step. This Actor is meant to be invisible when not in use, so to ensure it doesn't animate through the frames we created for the sprite, we also need to set it to [Animation Speed: None]. To ensure there won't be any part of it visible during load, I put a [Hide Actor] event in this actor's On Init section and left it set to "Self." A screenshot of scene settings from GB Studio.

Setting up the "Slide Move Actor" ### Variables and Values To store the information for each Tile, we need an assortment of variables to hold each value. You can start by creating the variables Tile 1 through Tile 16. Each of these represents the trigger the player can affect and what slide tile value is currently there. When we slide, we will update the current Tile to 0 and move the value of the current Tile to the value of the place it slid to.

Next, we will create six variables for positioning tile swaps: Tile X, Tile Y, Swap Tile X, Swap Tile Y, Move to Swap Tile X, and Move To Swap Tile Y. Tile X and Tile Y will provide the X and Y values of where the [Swap Tiles Var] event should swap within the puzzle scene, whereas Swap Tile X and Swap Tile Y are used for positioning the swaps on our tile reference background or Tilemap.

Positioning is important in a puzzle like this, so let's create one more variable: Current Tile. We will use this variable to set our current tile value and offer a base point to provide predefined values from a [Switch] event inside a custom script we will set up next. ### The Swap Tiles Var Plugin The improvements to GB Studio over the years, especially in version 4, can't be overstated. There have been so many quality-of-life improvements from handling scripts, color, and most recently, tile-swapping. While tile-swapping as a native feature is a blessing, it doesn't naturally offer a 16×16 swap mode like the Swap Tiles Plugin does.

For that reason, and ease of using variables instead of number values in the Var version of the plugin, we will be using this plugin in our project.

We will first create a **[Swap Tiles Var]** event that hides our unique tiles from the background. We don't want the player to see this, so we'll swap it out on Scene Initialization in our Puzzle scene. You'll notice that the Tileset values are represented by 0, as our blank Tile appears at 0,0 on our Tilemap background. We set the number to 16 tiles and each one's X and Y position will be the top left of the 16×16 square for each slot. A screenshot of a Switch event in GB Studio.

Swapping to the first (default) tile in our Tilemap sheet using the **[Swap Tiles Var]** event After putting this event into the On Init of the Puzzle scene, running the game will show all blank tiles. This is a good sign, but far from where we need to be to get our puzzle up and running. Let's dive into the logic behind the puzzle before we put everything together. ### Time for Logic When we plan out any complex scene with many moving parts and visuals, we first need to determine the best order and course of action before we get too far ahead of ourselves. Creating a plan for the logic used in a puzzle scene is essential and by breaking down what we have, need, and how to tie it all together, we can tackle this challenge.

Each of the 16 triggers already moves us around the puzzle space, which is great. To create the illusion of moving a tile from one space to another, we'll look at the breakdown of steps to determine what is missing: 1. Move the Slide Move Actor to the current trigger space (entering the trigger already does this) 2. Set the animation frame of the Slide Move Actor to the Current Tile variable value 3. Hide the Player, Show the Slide Move Actor 4. Swap the tiles behind the Slide Move Actor to blank 5. Move the Slide Move Actor to its destination Tile, get the new Tile location information for tile-swapping 6. Swap the tiles behind the Slide Move Actor to the correct Tile number 7. Hide the Slide Move Actor, Show the Player 8. Check for Win While these steps are pretty straightforward, there are many variables and events in-between to make it work. Ultimately, the largest focus is the illusion of a Tile sliding from its current space to the blank space which will take precedence over everything else. Now that we have an outline, you can begin the hardest part of designing the puzzle: coding it. ### Taming the Beast When creating the code for each trigger, we also check the available Tiles near them. For example, the bottom right trigger, Tile 16, is adjacent to Tile 12 and Tile 15. Some tiles will have up to four Tiles near them, so you must set up your script to check each Tile to see if it's False or has the value 0, representing the blank Tile.

An example of this written out logically would look like this: * If Tile 12 == False + Run the rest of the code * If Tile 15 == False + Run the rest of the code This check will be used for every option for every trigger (a total of 48!), so it's important to understand this concept as it's what makes the script work. A screenshot showing some of the scripting from the puzzle's code.

The beginning of the script for Tile 16's trigger The first part of the script is an **[If Variable Is False]** event that envelops the rest of it. In this example, if Tile 12 is the blank space, it can move on to the rest of the code. Within it is the **[Attach Script To Button]** event, which is immediately followed by an **[If Button Held]** event with all the directions set to on, to ensure no other directions are pressed before execution, or the player could move to a new spot and execute the code simultaneously. Note that you will put the main section of your script in the Else section of this event, as it only works if you check that they are not pressed. A screenshot from our script.

Locking, Setting, and Framing Start with a Script Lock to ensure players cannot move or otherwise during the script's execution. Next, set your Current Tile variable to the variable value of our current tile with a **[Variable Set To Value]** event, which in this example is Tile 16 (You need to click the # sign under the word Value to change it to \$ Variable instead). Use a **[Set Actor Animation Frame]** event, set the actor to our Slide Move Actor, change from value to variable like before, and set the variable accordingly. Hide and Show actor events in GB Studio.

Hide and Show is my favorite childhood game. Add two more events for the next part: **[Hide Actor]** and **[Show Actor]**. Set these to the Player and Slide Move Actor, respectively. Next, we need to create a custom script to use every time we want to get the tile values for swapping. Go to the SCRIPTS section on the left side of the GB Studio interface, and hit the + button to create a new one. Setting up the script to get the X and Y coordinates for our tile swapping.

Part of our Custom Script to get X & Y Values The first two events in this script are **[Set Variable To Value]**, converted this time to Properties. We are applying the Player's Tile X and Tile Y values to the Tile X and Tile Y variables. Next, create a **[Switch]** event with 15 options, set to the variable Current Tile. Within each numbered section, put two **[Set Variable To Value]** events, one for the X value in the Tilemap background where that number appears and one for the Y value, and set to the Swap Tile X and Swap Tile Y variables, respectively. Each (A) Button script will check this Current Tile variable and provide the X and Y coordinates from the Tilemap background to plug into the **[Swap Tiles Var]** event after this custom script. Return to your trigger script and add a **[Call Script]** event, setting it to the script we created. Configuring the Tile Swap plugin.

Calling the custom script, swapping the current Tile to blank, and setting some Tile values Before you apply the **[Swap Tiles Var]** event, you will subtract 1 from the Tile Y position with a **[Math Functions]** event. We did this before the swap as the top left of the 16×16 tile we will be swapping is the starting position for the tile-swapping code. Next, add the **[Swap Tiles Var]** event, assigning the Scene X and Scene Y to the Tile X and Tile Y variables we retrieved and altered. Finally, set the Tile # variable of the tile we're moving to as the current Tile # variable and set the current Tile # variable to zero (the value of the blank space in this puzzle), using modified **[Set Variable To Value]** events.

A screenshot from the Swap Tile plugin.

Moving and Swapping Add an [**Actor Move Relative**] event, plug in our Slide Move Actor, and set it to the direction of the tile we are moving to: * Up is {0,-2} * Down is {0,2} * Left is {-2,0} * Right is {2,0} Next, use a [**Store Actor Position In Variables**] event to store our newly moved actor's X and Y coordinates in variables Move To Swap Tile X and Move To Swap Tile Y. Repeat the previous action of offsetting the Y coordinate with a [**Math Functions**] event, this time subtracting 1 from the Move To Swap Tile Y variable to prepare for our next event, the [**Swap Tiles Var**]. Change the Scene X and Scene Y to the Move To Swap Tile X and Move To Swap Tile Y variables you retrieved and altered, and then set the Tileset X and Tileset Y to the Swap Tile X and Swap Tile Y variables you got when you called the Get X & Y Tile Values for Swaps script and then edited the Y value. This will swap the tile you moved to behind the actor before you hide it. Finally, use [**Hide Actor**] on the Slide Move Actor and [**Show Actor**] on the Player to reveal the player's cursor.

To finish everything we've done, you will create another custom script named Check For Win. This one will check to see if all the tiles match their appropriate values, and if so, you solved the puzzle. Start with creating an [**If Math Expression**] event, using the following expression within:

```
($Tile 1 == 1) && ($Tile 2 == 2) && ($Tile 3 == 3) && ($Tile 4 == 4) && ($Tile 5 == 5) && ($
```

A screenshot of the script that checks if the game has been won.

Figure 3: A screenshot of the script that checks if the game has been won.

Checking for a win state For a true expression, it will hide the player with a [**Hide Actor**] event and display an appropriate [**Display Dialogue**] event with something along the lines of "You Solved The Puzzle!" Return to the trigger script you were on and use a [**Call Script**] event to call the Check For Win script. Add a [**Script Unlock**] event and finally, a [**Remove Button Script**] set to the (A) button. Now that this script is assembled, it's time to spread it and alter it for each of the triggers on the grid. ### Scripts in Bloom Usually, when you reuse a set of events repeatedly, you use a custom script, however, values within certain events of this script are altered for each option. Let's highlight what you need to change in each version of this script: * The Tile # variable at the top, seeing if the tile you intend to move to is false/empty/0 or not * The Current Tile = Tile # (only needed to be changed once for each trigger, then copied to the script for the other directional option. Example: Tile 12 and Tile 15 – both have a Current Tile value of Tile 16) * Set the animation frame of the Slide Move Actor to the current Tile # (Example: Tile 16) * After the first tile swap to blank behind the actor, you made the tile you are moving the same value as the current tile (Example: Tile 12 = Tile 16) * The current tile is set to 0 (Example: Tile 16 = 0) * The [**Actor Move Relative**] event which is dependent on which direction the intended tile is (Example: Move {0,-2}) A screenshot of how the tile movement events are set up.

Expanded are the events we are altering for every intended tile that can be moved to. Since only one Tile # variable can be set to 0, you don't need to include the code in nested Else, as most scripts do. Instead, you will just paste the script you filled out, leaving the Current Tile the same, for however many directions are available for each one. The setup of our Attach Script to Button event.

What it looks like with the [Attach Script To Button] events minimized for the example trigger (16) To set up the rest of the grid, you set up the possible tiles that can be traveled to from each trigger. Here's a list of each option for each grid position: 1. Tile 2, Tile 5 2. Tile 1, Tile 3, Tile 6 3. Tile 2, Tile 4, Tile 7 4. Tile 3, Tile 8 5. Tile 1, Tile 6, Tile 9 6. Tile 2, Tile 5, Tile 7, Tile 10 7. Tile 3, Tile 6, Tile 8, Tile 11 8. Tile 4, Tile 7, Tile 12 9. Tile 5, Tile 10, Tile 13 10. Tile 6, Tile 9, Tile 11, Tile 14 11. Tile 7, Tile 10, Tile 12, Tile 15 12. Tile 8, Tile 11, Tile 16 13. Tile 9, Tile 14 14. Tile 10, Tile 13, Tile 15 15. Tile 11, Tile 14, Tile 16 16. Tile 12, Tile 15 Once all of those are set up, you will need a way to test everything is working properly. It's time to set up some debug options. ### Debug is love, debug is life Debugging, or adding sample code that tests things without actually performing those actions, is essential for testing any game, especially a puzzle game. Another feature within the latest versions of GB Studio is the ability to watch variables as they are affected by gameplay. Let's set up both so we can test out everything you've coded so far.

First, make an event group called "Debug" and put it into the On Init space of the Scene you are working in. Next, use an array of 16 [Variable Set To Value] events. Set each Tile # to its respective value, except 16, which is set to 0, like this: A screenshot of the debug script setup.

Setting all our Tile # variables to their expected values We will reuse our [Swap Tiles Var] event from the beginning and fill in the locations of all the numbered tiles from our Tile Swaps scene. Here are the Tileset X and Tileset Y values for each: 1. 2, 0 2. 4, 0 3. 6, 0 4. 8, 0 5. 10, 0 6. 12, 0 7. 14, 0 8. 16, 0 9. 18, 0 10. 0, 2 11. 2, 2 12. 4, 2 13. 6, 2 14. 8, 2 15. 10, 2 16. 0, 0 Now, we are ready to run the scene with debugging on. Head to the Game menu at the top left of GB Studio and select "Run With Debugging." A view of the debug screen for the game.

All of the variables we click the star next to will show up in "Watched" You'll see a list of variables behind the running game. Hit the star next to all the Tile # variables, and click on "Watched" to the right where it currently says "All." As you play around with the puzzle, you'll notice the values switching around in the Debugger menu. This will show you how the variables are working or whether they are not working correctly. Move the blank space around the entire playing field, the values should move along with it. If not, you might have messed up some code somewhere.

Hopefully, you have now confirmed that all of your code is working. Close the game window and open it again, move the 15th tile to the 16th spot, and revert

it. You should activate the “Win” state, and receive the message you set up earlier in the Check For Win script.

If so, congratulations, you have a working sliding puzzle! If it’s still not working, retread your steps and determine where you went wrong. Use the debugger for additional clues to check what values you are getting on your variables as you move around and use the (A) button. ### Choose your Setup You have a couple of options for how to set this puzzle up. You could convert the tiles to a picture and change the graphics from numbers to parts of the whole picture, or you could jumble up the numbers and have the player try to slide it back to the winning setup. For the sake of simplicity, let’s assume you merely want to jumble the numbers up.

Let’s return to our “Debug” section that we set up earlier. Rename it to “Puzzle Set Up” and then figure out what numbers you want where and change the Tile # values to match. Then, edit your [Swap Tiles Var] event inside to match the positions accordingly. You can also disable the blank Background Swap [Swap Tiles Var] event we used above our event group as we no longer need to swap the entire puzzle to blank spaces. Finally, test your new puzzle and see if you can solve it yourself. If so, let someone else have a try and see if they can figure it out as well!

In **Part 3**, we will set up random puzzle generation and a menu system, and add some additional visual flair to our Sliding Tile Puzzle.

This article is part of a series: Let’s Program a Puzzle Game! Pearacidic Pearacidic

Writer, Game Designer, Engine Bender, Retro Enthusiast. Co-creator of the In The Dark series and Grimace’s Birthday. (He/Him)

pearacidicgames.com/ ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### Related

Let's Program a Puzzle Game! - Part 3

Random Puzzle Generation

Welcome back! Now that our puzzle is functioning and you can beat it, we should add some quality-of-life improvements. Since nothing happens after getting the win state, we need to add replayability by offering a new puzzle. To do that,

we’ ll add random puzzle generation to the code. Beyond that, we need a menu system to give the player a choice to start a new puzzle when finished. These additions take our puzzle from a simple solution to a fully realized game.

Let’ s begin by creating a new custom script named Random Tile Assignments. We will use this script to grab a randomly generated number (2 through 15) and make the puzzle in sequence from whatever was picked.

Add two [**Variable Set To Value**] events to the top of the script, setting the first one with the Tile 16 variable equal to 0. The second one will be switched from “number” to “variable” and set Tile 1 equal to a new variable Last Tile 1. This variable will be set to ensure the player doesn’ t see a repeated puzzle from the last one generated.

Next, we will add a [**Loop While**]event, setting the condition to Tile 1 and the equals to Last Tile 1. You need to add a [**Math Functions**] event inside this event, as this allows us to continue running a random number function until we arrive at a different number than last time. Set the variable to Tile 1, the Value to “Random” , the Min value to 2, and the Max value to 15. Usually, we would worry about needing a random set that is either binary or divisible by 8, however, since all we are doing is looking for a different number, this should suffice.

Add an [**Event Group**] event, and make an array of [**Evaluate Math Expression**] events inside. Within each event, we’ ll set the Variable to the Tile # variables 2 through 15 and the Expression to “\$Tile # + 1,” where the Tile # is the previous # in the sequence. For example, Tile 2’ s Expression would read “\$Tile 1 + 1.” This adds 1 to the Tile 1 value randomly generated earlier, and the following events would continue the sequence, filling in the values from the top left of the puzzle space.

Starting the Random Tile Assignments Custom Script Finally, add a series of [**If Math Expression**] events, each containing \$Tile # > 15, where the #’ s are 2 through 15 again. Within, we’ ll use a [**Math Functions**] event in each one to subtract 15 from each respective Tile # variable. This checks to see if the value of that variable is over 15 (our highest displayed tile), and subtracts 15 from it, if so.

Checking for values over 15, with Tile 2 as an example For us to use this newly minted custom script, we’ re going to need another script to swap the tiles we determined with our random pick. Something you must always concern yourself with during GB Studio development is making scripts too big, and for this reason, we’ ll split this additional code that handles the tile swaps into a separate script. ## Random Assigned Tile Swaps Create your custom script titled Random Assigned Tile Swaps and begin by creating 15 [**Event Group**] events, renaming each from 1 to 15. Inside each one, use a [**Variable Set To Value**] event to set the Current Tile variable to the Tile # variable that matches each event group. For example, in event group 1, it would be \$Tile 1. Next, we’ ll copy over the Current Tile Switch we made for the Get X & Y Tile Values for Swaps custom

script. We’ ll finish each with a **[Swap Tiles Var]** event, with the Scene X and Scene Y values found in the Background Swap script in Part 2 from our Scene’ s On Init code.

Adding code to the Random Assigned Tile Swaps script Now that we have both of these custom scripts together, we’ re going to use them in multiple areas to set up the rest of our project. We’ ll need to use them to initialize the puzzle scene as well as within a pause menu. **## Start The Shuffle** When we start the puzzle scene, we want to give the player an appearance of control over the random puzzle generation. To do this, we’ ll add two new sprites:

Press Start Sprite

Shuffling Sprite Go ahead and add a blank frame to the end of each sprite in the Sprites section of GB Studio and make sure the Animation Type is set to Fixed Direction. You will then take these sprites and apply them to the Sprite Sheets of two new actors in the scene, “Shuffling” and “Press Start.” Go into the On Init script for Shuffling and add a **[Hide Actor]** event, leaving it set to “Self.” Set both of these to an Animation Speed of 3. Place them in the scene as shown here:

The Puzzle scene with the new actors added With our shiny new actors set up, it’ s finally time to piece together our scene’ s On Init script and finish our “Check For Win” script with an additional context menu. **## Ending With Initialization**, and the *Check It’ s* a rather unorthodox place to finish, but we’ ll go back to the beginning of your setup in the On Init script where we previously added the DEBUG Section (or you changed it to Puzzle Set Up to test a custom version of the puzzle) and if you haven’ t disabled your Background Swap **[Swap Tiles Var]** event, disable that now. We’ re going to disable the DEBUG event group and add a **[Hide Actor]** event, set to Player after it.

To help organize the next section, create yet another **[Event Group]** event and label it “Puzzle Initialization.” Inside, add a **[Pause Script Until Button Pressed]** event and set it to the Start button to keep players from being able to move around before starting the puzzle. Next is an **[If Button Pressed]** script that will encompass the rest of the code. **### Start Button Code** Starting with a **[Script Lock]** event, we’ ll copy or drag & drop the Background Swap event from above and enable it here. It’ s a bit of redundancy, but either add or copy the **[Hide Actor]** event, set to Player. Next, add a **[Seed Random Number Generator]** to help with the random number selection for puzzle generation in our custom script, which will randomize the seed based on the frame data from the scene. Use another **[Hide Actor]** event, this time set to our Press Start actor, followed by a **[Show Actor]** event set to the Shuffling actor. Add a **[Call Script]** event set to the Random Tile Assignments script and a **[Wait]** event set to 0.5 seconds or whatever you prefer. This wait is not necessary for anything other than showing the Shuffling actor for a while to let the player know something is happening. Call the Random Assigned Tile Swaps script with another **[Call Script]** event, then **[Hide Actor]** to hide our Shuffling actor now

that the puzzle has been shown. We’ ll move our player to 12,12 with a [**Set Actor Position**] event, [**Show Actor**] to show our player, and finish with a [**Script Unlock**] to give the player control again.

Initializing the puzzle To finish off our Puzzle Initialization, add a [**Attach Script To Button**] and set it to Start. This will serve as a pause menu button. Copy or create a [**Hide Actor**] event set to Player, then add a [**Display Dialogue**] event.

In menus, I prefer to use the Variable Width font, which can be selected if you type an “!” in the Dialogue box followed by the letter “f,” and select Font: GBS Variable Width from the options. Next, type another “!,” but this time follow it with an “s,” as it will give you “Speed” options for how fast to render the text. Set this to Instant so it appears more like a menu than dialog from a character. These two additions should appear as “FS” in your box. Finish the text in the box by adding “New Puzzle?” and then use spaces to move it over, citing the preview window by hovering your mouse over the event, to center the text to the box. Switch over to the Layout Tab in the [**Display Dialogue**]event, and change only the Min Height, Max Height, and Scroll Height to 3, leaving the rest of the options default.

Create a new variable named Menu Option and set it as the Set Variable in a [**Display Menu**] event. Set to 1 if Yes, and 0 if No. Check off the “Last option sets to ‘0’ ” option, and change the Layout to Menu. To finish this menu, we need to check if the Menu Option was set to Yes or No, True or False, or even 1 or 0. Add a [**If Variable Is “True”**] event, and set it to Menu Option. Inside, we are going to copy the exact contents of our Start Button Code, save for one small change: remove the [**Seed Random Number Generator**] event. Finally, in the Else section of the script, add a [**Show Actor**] event and set it to Player.

Scripting our pause menu set to the Start button With our initialization script done, we only need to add the final menu to the Check For Win script to complete this puzzle game tutorial. Luckily for us, it’ s simply a matter of copying & pasting the exact code we just finished and altering our existing [**Display Dialogue**] event. Use the shift key, and click on each event to highlight both the [**Display Menu**] and [**If Variable Is “True”**] events. Pull down the context menu and select ‘Copy Events.’ Then, in our Check For Win script, use either the Alt or Option key and click where it said “+ Add Event” before, as it should now say “[+] Paste Event.”

Copying and pasting the events from the Pause Menu script Once you paste these events, when the player solves the puzzle, it will prompt them with the existing box stating “You Solved The Puzzle!” , add an extra line below that that asks them if they want to “Play Again?” This will lead them into the menu we have set up, creating a loop back into starting a new puzzle.

Our final “Win” dialogue event Now that we have finished all of our steps, we have a puzzle game with random puzzles, a visual indication to the player what button to press to begin, and a loading sequence to inform them when the game

is loading. There is plenty more that can be done to customize our puzzle, but I will leave that for you to decide.

I can't wait to see what you create with this new knowledge, and even though this is just a dip into the ocean that is puzzle games, there are plenty of skills within this lesson that can be applied to other aspects of design, especially with UI and menu design elements.

If you haven't yet, join the official GB Studio Discord and share what you've done with our community! Also, if you want, you can check out both of my puzzle games, *In The Dark* and *In The Dark 2*.

Good luck and happy puzzling!

This article is part of a series: *Let's Program a Puzzle Game!* Pearacidic Pearacidic

Writer, Game Designer, Engine Bender, Retro Enthusiast. Co-creator of the *In The Dark* series and *Grimace's Birthday*. (He/Him)

pearacidicgames.com/ ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### *Related*

IX. Debugging & Testing

- How to Debug Your Game
- Testing Your Game on Different Hardware

Testing Your Game on Different Hardware

When you're creating a game, you obviously are going to test it, to look for bugs, see if mechanics work out and how the game feels.

If you're using GB Studio, you'll most likely use the built-in emulator the most for that purpose. That's a good start, but only scratches the surface.

Depending on the game you're creating, it is very important to test it on as many devices, systems and emulators as possible, because there are more obstacles to tackle than "it just doesn't run."

Even if you build and design your game with emulation on computer displays specifically in mind, players may download your rom file to play and experience it

on their hardware, be it the original Game Boy with the help of flash cartridges, modern retro handhelds, or other consoles.

No matter what you do, you can't please every player and making the game perfect for every device and edge case would be a never ending money and time sink, but it's a good start toward making your game work for as many devices possible, or at least being aware of the situations where your game doesn't live up to its full potential due to its limitations.

In this article I'll cover why it makes sense to test your game on multiple devices or emulators, and point out the problems you might consider addressing or keeping in mind. # Screen Sizes You'll most likely use some sort of computer/laptop to develop your game (if not, let us know, because that sounds wild and we're interested to hear about it). The screen you're using and the emulator window popping up to test your game will in most situations already be larger than the original hardware screen.

You have to keep that fact in mind when creating the pixel art for your game. The limitations of the Game Boy are already pretty steep, but if you're designing sprites that are barely identifiable on the bigger screen, then players will have a harder time on their original hardware. A photograph of the Anbernic RG405M console.

Typical modern retro handheld emulating Game Boy games (Anbernic RG405M with a 4 inch screen) # Colors This topic should be taken care of outside of the spectrum of testing on multiple devices, as strong contrasts in general are better to see, especially for people with color blindness or other disabilities that makes it hard to differentiate certain colors.

But the same principle applies when having different devices in mind. For example, if you're building a game with fast paced movements and you designed the levels with very detailed pixel art and less contrast in it, movements will start to blur, which makes it hard to see oncoming obstacles or enemies on an original Game Boy. This blurriness is less noticeable or completely invisible on newer devices with modern IPS screens. A photograph of Super Mario Land 2 being played on an original gameboy. The game is very hard to see due to movement blur.

Super Mario Land 2 on the original Game Boy DMG-01 – It's hard to tell what is an enemy, what is an obstacle, and what is the background, because of the movement blur # Input Lag Game Boy games are known to be a bit slower paced. Mario games, *the* prime example of a typical Nintendo platformer, have slow-walking acceleration, and often feel like running on ice. Even if you're at full speed, it's not *that* fast compared to modern platformer titles on other consoles or PC. Still, precise input is required to let Mario land where you want him to land.

Input lag in games where precise input is needed to succeed can be very frustrating when played on the wrong setup. On original hardware, input lag isn't a problem

to worry about. Most of the time input lag is introduced through less-than-ideal emulation or the machine running the emulator isn't powerful enough for it to run smoothly.

I'm currently creating an action platformer with very high movement speeds for the Game Boy, which requires precise and quick input. On original hardware the input is flawless. On my laptop through the GB Studio emulator, the input is okayish, depending on what is running simultaneously in the background. But in my playtests with other people I found out that input lag can also be introduced through things like wireless keyboards or controllers!

All playtesters who had wireless keyboards for their computers (where they were testing the game through emulation, just like on itch.io) got back to me and said that the input felt a bit off and they had a hard (and maybe frustrating) time playing my game because of that.

Only when I asked them to use a wired keyboard or another method like playing on original hardware was the problem fixed, and playtesters felt significant differences in input lag.

These are things to have in mind when you plan to bring such a game to platforms like Steam, where users will experience the game through an emulation wrapper, and many players own wireless keyboards and controllers that will introduce input lag. A photograph of a 3D printed handheld game console.

Self-3D-printed and built handheld. You never know on which hardware players will run your game # Imprecise Input Another thing to consider is how well can your game be played on devices that are controlled via touchscreen?

The standard web export on GB Studio offers the game wrapped in an emulator, served to be played via web, either with a keyboard or via touch controls on devices like smartphones.

If you're uploading your game on platforms like itch.io to be played there, then this is one of the first ways players will be introduced to your game.

Playing a game like Super Mario Land would already be a challenge with touch controls, where your fingers can't feel if they're touching the right button, or any buttons at all. Now imagine having to play a fast-paced action platformer with very precise inputs and one-hit-kill mechanics. This is where frustration will be born! # Multi Input An emulator on the computer offers another thing, that, if not kept in mind, could cause game-breaking bugs or mechanics: that you can press multiple keys at the same time.

If you're building a mechanic that is being performed by pressing Up and Down simultaneously, then it could work on your computer, maybe even on touch screen devices, but it will not be possible on original hardware, because of the way the input is built on the Game Boy.

I know that this information is a bit of a "duh?" moment, since everyone who grew up with or owned a Game Boy knows this. But there are also plenty of

younger developers who were never lucky enough to get their hands on an actual Game Boy DMG-01, and that's why I think this is a good thing to know, as you could in fact make a mechanic like "Pressing Up and Down at the same time" . A photograph of a Game Boy color that has been modded with an IPS screen, improving its visual quality.

Game Boy Color with IPS screen mod # Computing Power & Emulation While modern devices like smartphones or the Analogue Pocket surely have enough computing power to handle a heavy Game Boy game, there can be certain mechanics, scripts, and other aspects in the game that can bring even those powerful devices to their knees.

A poorly made emulator can have a slowdown while walking through a busy part of your game handling multiple actors and effects. Or a kernel panic error could occur on the Analogue Pocket where it would be running safely on your original Game Boy.

So testing on multiple devices and emulators is a good idea to check if your game could have issues in certain situations. ## **What can you test on easily?** I know not everyone can just buy a handful of different retro handhelds, original Game Boy (Colors), and other devices & systems that possibly can run Game Boy games. But you probably can test on: * the built-in GB Studio emulator * the web export version of GB Studio on your computer (keyboard, controller) * the web export version on your smartphone (touchscreen testing) You can also always ask the community discord for people to test your game on different devices that you don't have. It should at least be tested on: * Original Game Boy * Original Game Boy Color * Analogue Pocket * Super Game Boy (SNES) * Some retro handheld devices (Anbernic, Powkiddy, Retroid, you name it) Martin Cremerius-Gauer

Martin Cremerius-Gauer

Martin first went into making GB Studio games with his playable portfolio of his own personal website. He is generally interested in a very broad band of topics, and adds new things to the skill list all the time. Being born in the late 80s, the original Game Boy DMG-01 was his first gaming console, and so he kind of has a nostalgic romance with everything around it.

2ndlwgames.com ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### *Related*

X. Accessibility

- Breaking Barriers: Develop GB Games For Gamers With Visual Impairments
- Breaking Barriers: How To Develop Accessible GB Games For Gamers With Disabilities

Breaking Barriers: Develop GB Games For Gamers With Visual Impairments

As I already mentioned in the starting article of this series, my reality is a different one than yours, since I have a slight form of dyschromatopsia.

Since it's very minor, I usually don't notice the effect of this disability in my daily life, or while gaming, thus I can usually enjoy all games without any problems. Though, other people can have a harder time playing games because of very low vision (even with the use of glasses) caused by several visual impairments, color blindness, or even near-complete or complete blindness, making it a lesser experience. In this article I want to showcase some solutions to offer better support on your next Game Boy and Game Boy Color game for people with visual impairments.

Of course it's hard, or for some types of games straight up impossible to consider all of the mentioned methods. Thus the following list is by no means a fixed set of rules to go by with every game you create. But I think raising awareness for this topic is important and it may lead to some sensible design decisions in your future games. ## 1. High-Contrast Sprite and Background Design **Maximize Contrast:**Make sure to use the darkest and lightest colors to differentiate between important game elements (e.g., character sprites vs. background). Prioritize the visibility of interactable objects.

Avoid Busy Backgrounds:Keep backgrounds simple and uncluttered to prevent confusion with key gameplay elements. Use solid colors or simple patterns behind important objects or text to maintain clarity.

Edge Outlines:Give important sprites (like characters, enemies, items) clear outlines in either black or high-contrast colors on GBC. This helps them stand out against the background. ## 2. Colorblind-Friendly Design (GBC) **Patterns and Symbols:**Since the GBC offers, well, color, ensure critical gameplay elements (such as enemies or items) aren't solely differentiated by color. Use patterns, shapes, or other visual indicators when possible to make out the difference.

Limited Palette:Choose contrasting color palettes that work well for all players, including those with colorblindness. Test your palette for visibility issues like protanopia or deuteranopia by sticking to universally distinguishable colors (e.g., avoid red-green contrast).

Use tools to check your colors: * <https://color.review> * <https://color->

blindness.com/coblis-color-blindness-simulator ## 3. Sound as Feedback **Audio Cues:** Use sound effects and music strategically. Since the Game Boy has no text-to-speech capability, audio feedback becomes crucial. Assign unique sound effects to actions like collecting items, taking damage, or activating switches, so players don't rely solely on visual feedback.

Interactive Elements: When a player interacts with an object (like a door or switch), provide immediate sound feedback so they know the interaction succeeded. ## 4. Clear Gameplay Mechanics **Slow-Paced Mechanics:** Avoid fast-paced, twitchy gameplay that requires split-second visual responses. Instead, give players more time to react to what's happening on screen. This helps players with low vision or slower visual processing. The low refresh rate of the original Game Boy with its screen tearing already forces us developers to not go wild with too fast things happening.

Minimal Reliance on Visual Precision: Design levels that do not require pinpoint accuracy. For example, wide hitboxes for characters and objects can ensure that visually impaired players don't need perfect hand-eye coordination to perform essential actions like jumping or attacking.

Visual Indicators for Game States: When something important happens (e.g., player low on health, level up, or a boss fight is about to start), use both visual and audio signals to make these states noticeable. ## 5. Text and Dialogue Accessibility / Font Design **Readable Fonts:** Avoid using overly stylized fonts that could be hard to discern on the small screen the Game Boy has. Fonts without serifs are usually better to read.

Spacing: Ensure enough spacing between letters and lines of text when choosing or making the font. Crowded text can be difficult to decipher, especially for players with low vision.

Slow Scrolling Text: If you're using text-based dialogue or storytelling, allow text to appear on screen slowly or give players control over text speed. Players with visual impairments may need more time to read.

Sound Cues for Text: Add an audio cue when new text appears to help visually impaired players keep track of the narrative or important instructions. ## 6. Menu and UI Design **Simple, Clean Menus:** Keep the user interface (UI) and menus straightforward, with large text and distinct icons.

Highlight Selection: Ensure the currently selected menu option is clearly highlighted. Use a bold outline, high-contrast colors, or even a simple animation (like a blinking selection box) to show what is selected. ## 7. Test on Real Hardware If possible, playtest on actual hardware. Emulators often display Game Boy graphics and colors in a way that doesn't reflect how they appear on real hardware, especially in terms of brightness and clarity. Make sure to test your game on a real Game Boy or GBC and other handhelds to ensure that the visuals are clear and readable on all screens. ## Verdict By focusing on these tips you can create a more accessible experience even within the limitations of the Game Boy and Game Boy Color hardware. Playtesting with visually

impaired players will also be key to refining your approach and ensuring that your game is enjoyable for a broader audience. On top, many of these tips are actually the base of a good and polished game, so it makes sense to have them in mind anyway.

Let's try to think outside our own scope and make sure to include more players by designing with accessibility in mind.

This article is part of a series: [Breaking Barriers](#) Martin Cremerius-Gauer

Martin Cremerius-Gauer

Martin first went into making GB Studio games with his playable portfolio of his own personal website. He is generally interested in a very broad band of topics, and adds new things to the skill list all the time. Being born in the late 80s, the original Game Boy DMG-01 was his first gaming console, and so he kind of has a nostalgic romance with everything around it.

2ndlawnames.com ### Share this:

- [Click to share on Bluesky](#) (Opens in new window) Bluesky
- [Click to share on X](#) (Opens in new window) X
- [Click to share on Facebook](#) (Opens in new window) Facebook
- [Click to share on Mastodon](#) (Opens in new window) Mastodon
- [Click to share on LinkedIn](#) (Opens in new window) LinkedIn
- [Click to share on Reddit](#) (Opens in new window) Reddit ### *Related*

Breaking Barriers: How To Develop Accessible GB Games For Gamers With Disabilities

In the article “The State of Accessibility in Gaming in 2022,” Cam Waller states about 20% of the estimated 2.8 billion gamers around the world have some kind of disability, facing certain barriers and challenges to play video games. ## Your Reality May Vary I'm absolutely fascinated by the fact that almost everyone's reality is different. But what do I mean by this?

If you have the use of your eyes, then what you can see, in addition to your other senses, is in fact your own unique experience of reality meaning everyone has their own way of experiencing the world.

Let me give you an example: I have a very slight form of dyschromatopsia, which means I'm not as good at differentiating certain red and green color nuances from each other than people without dyschromatopsia. A picture of an Ishihara test. Dots in shades of red and green arranged in a circle, with a 7 visible to non-colorblind viewers.

(Image: Ishihara test) For instance, on this Ishihara test, I have to zoom-in to see a 7. For this article I even had to ask my fiancée to make sure there is in fact a 7. For her, the 7 in this image is very clear and readable, even from a distance.

To me, the further I hold this image away from me, the less I can even make out that there is a number or something at all.

Her reality is different to the reality I experience. Surely I can still fully function in this world, and I would say, unless it's an Ishihara test, I don't even notice I have this disability. But at the end of the day, I can't see certain things as clearly as other people can.

This is true for many people on this planet, often with more serious disabilities which make their daily lives a bit harder to navigate, to say the least.

Focusing purely on video games, whether someone experiences color blindness, is missing an arm, or difficulty with memory, many gamers must endure extra hurdles, or in some cases are entirely unable to play games as designed. ## Accessibility of the Game Boy (Color) It's common for modern consoles, such as the Nintendo Switch or the Valve Steam Deck, to have accessibility options to adjust settings to help players with disabilities.

There even exist external accessories to help in certain situations to be able to play games.

And although modern ways of playing Game Boy games, like emulation on modern hardware, or new retro gaming consoles offer certain features like slowing down the game's speed, altering display color settings, or introduce save states, the amount of options that the console a game is running on can change inside of the games themselves are fairly limited.

That's where we as developers come into play! ## Breaking Barriers Articles: In this article series, I want to raise awareness of some categories of disabilities, and explain what we as developers of Game Boy (Color) games can do to make our games accessible to more people around the world. In each article, we'll discuss tips, tricks, and mechanics we can use to achieve this goal!

Let me shortly introduce you to the categories in this series: ### Visual Impairments Link: Breaking Barriers: Visual Impairments * Including, but not limited to low vision, color blindness, and blindness. ### Auditory Impairments Link: Breaking Barriers: Auditory Impairments (Coming Soon) * Deafness or hardness of hearing. ### Mobility Impairments & Motor Disabilities Link: Breaking Barriers Mobility Impairments & Motor Disabilities (Coming Soon) * Difficulty with physical movement, including limited range of motion, paralysis, or missing limbs. * Difficulty with fine motor skills, coordination, or dexterity. ### Cognitive Disabilities & Neurological Disabilities Link: Breaking Barriers: Cognitive Disabilities & Neurological Disabilities (Coming Soon) * Learning disabilities, memory impairments, or difficulties with processing information. * Conditions such as epilepsy, which may be triggered by certain visual effects or patterns.

This article is part of a series: Breaking Barriers Martin Cremerius-Gauer
Martin Cremerius-Gauer

Martin first went into making GB Studio games with his playable portfolio of his own personal website. He is generally interested in a very broad band of topics, and adds new things to the skill list all the time. Being born in the late 80s, the original Game Boy DMG-01 was his first gaming console, and so he kind of has a nostalgic romance with everything around it.

2ndlbgames.com ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
 - Click to share on X (Opens in new window) X
 - Click to share on Facebook (Opens in new window) Facebook
 - Click to share on Mastodon (Opens in new window) Mastodon
 - Click to share on LinkedIn (Opens in new window) LinkedIn
 - Click to share on Reddit (Opens in new window) Reddit ### *Related*
-

XI. Publishing & Community

- Basics: Switches
- So You Want To Crowdfund? Part 1
- So You Want To Crowdfund? Part 2
- So You Want To Crowdfund? Part 3
- So You Want To Crowdfund? Part 4
- Using Itch To Host Your Games
- Version Control and GB Studio: Getting Started
- Version Control and GB Studio: Reverting Changes
- Version Control and GB Studio: Using GitHub Desktop

Basics: Switches

In the “Basics” series, we’ ll be going over all the technical foundations needed to start building a GB Studio game. If you’ re a GB Studio Beginner, then these articles are for you. Seasoned GB Studio devs may still find different ways to tackle challenges with these articles, so it should be worth the read. As you develop your game and work with variables, you may have encountered situations where you’ ll need to check a variable for specific values. In some cases, the **[Compare Variable to Value]** event works quite well – but when you need to check multiple values, you may find yourself stacking many Compare Variable events. Before long, your scripts start looking like this:

If, then, If, then, If, then... While there’ s nothing wrong with stacking multiple compare events, this can quickly get out of hand. The more comparisons you need to make for specific values, the longer your script will become, creating more room for error and making your scripts harder to review. Luckily, an event exists specifically for this purpose! # Switches A switch is a single event that checks the value of a variable, and then runs a script when the variable matches an specified value. You can think of a switch as a way to combine multiple **[If**

Variable Equal to] events within a single event. Before getting into use cases, let's dissect a Switch.

- Every Switch is connected to a single Variable.
- A switch can check for **up to 16** specific values, which are set by its **Number of Options** sections.
 - These 16 values start at 1 and increase in ascending order as you add additional options, but you are able to change the values to any number within variable limits (0-255), and place them in any order.
 - A Switch also has an “Else” section, which you can use to check for values beyond these 16 options. Like other events, you can also disable the Else if you don't need it.
- After setting your Variable and the number of Options, you will see sections labeled “When: X” . These are your Compare Events. Set a **Value**, and when the Variable is **equal to** that value, the switch will run the script contained in that group.
 - Since switches are looking for an “Equal to” value, it will “break” and not check other values after it executes a script for a matched value. Using the “Else” section is useful for these situations. # Why Use a Switch? Aside from consolidating your script, Switches are excellent events to use for common situations where you'll be frequently checking a variable for *specific* values. Simplify Menu Options The [Text: Display Menu] event can have a maximum of 8 options, and each option is set with a connected variable to a specific value between 1 – 8 (or 0, if using that option). Using a switch with the same variable as your menu allows you to control all 8 options with a single event! For example:

Rotating Dialogue If you'd like your NPCs to rotate through dialogue options, you can use one of their **Local Variables** to count the number of times you've spoken to them. Then, using a switch connected to that variable, you can create different dialogue options.

Whenever you interact with this NPC, it Increments its Local 0 variable by 1 (if you remember, variables start at 0 until set, so this will immediately set our variable to 1 the first time we talk to the NPC). Then, we use a Switch to check the same Local 0 variable. The first time we speak to the NPC, the variable will be 1, so it displays the first dialogue. The second time, the variable is increased again, so it will show the 2nd option. Each time after that, because we only have 2 options to check, it will run the dialogue in the “Else” section. You could expand this even more by adding more dialogue options, or by resetting the variable when you've reached your else. Using Switches for dialogue can also be a great way to create branching dialogue. Creating switches based on the answers a player provides, or if other conditions are met, can allow you to create more engaging conversations with your characters. ## Scene Changes When changing scenes, you can use a Switch to create multiple options for where the player is sent. This can be as simple as using a switch to check if a player has

found a key or completed an objective (much like a True/False check), or even randomizing the next room by using the Switch in combination with a **[Math: Set Variable]** to Random event with the range being your switch outcomes.

Controlling States Much like controlling the dialogue an Actor displays, you can control their behavior using Switches. In our article on Enemy Movement, we cover an example on how to use a Switch in combination with other events to create an enemy movement pattern. You can also use switches to check for other states on a scene's Init. In this example, we can change our player's sprite between 2 different styles. We can track what "Look" our player selected using a Global Variable, and then use a switch to set the player's sprite sheet. To guarantee that the change persists between scenes, we can put the same switch on our Scene Inits, so this way when we change rooms, we still look the same. You can even have this happen on a button press – in an adventure game, you may want your player to cycle through different weapons, but keep the attack action assigned to the same button. In the example below, the player can rotate their Blaster's attack mode between a Close Attack and a Projectile Attack by pressing the Select Button. When the player presses the Attack button, (B), the script checks the variable for a mode using a switch, and performs the associated attack.

Wrapping Up Switches are powerful events that can be helpful for consolidating your scripts, but they're not a complete replacement for the Compare Variable event. Here are a few final tips for using switches:

- * Switches only check for "Equal To" values, you'll need to use Compare Variable events for "If Greater than" or "If Less Than" comparisons.
- * Switches can only have 16 options and an Else – but you can still look for more values on the same variable by placing another switch in your Else section.
- * Switch values do not need to be 1 – 16. Your "When: X" values can be any value between 0 – 255. I do recommend keeping them in a certain order just to make it easier to review.
- * Combining switches within switches can be a great way to create varied Actor behavior!

Practice Challenge: Using what you've learned about Switches, create an Elevator that takes your player between multiple scenes. Dan Enders

Dan Enders

Friendly Neighborhood Game Designer, Podcaster, and (ex)Librarian (he/him)
GB Studio Games | Tabletop Games **###** Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit **### Related**

So You Want To Crowdfund? Part 1

The Warp Coin Catastrophe was the first game made with GB Studio to be

successfully funded via a Kickstarter. In 2019, after a 21 day campaign, the project was funded at 151% of it's original goal which amounted to a little over \$9,000 CAD. What began as a small idea launched me into a much larger community of homebrew developers and crowdfunding project creators who shared their experience and wisdom with me. I thought it would be worthwhile to now share my experiences and perspective on the whole endeavor through the lens of a GB Studio developer. Perhaps you were considering trying to obtain funding for your own game, or maybe you just wondered what the whole process is like, these series of articles will bring through the whole journey from planning to shipping and hopefully answer your questions. So with that in mind, let's start at the beginning, which is the first question you should be asking: # To Crowdfund Or Not Crowdfund Crowdfunding has become a legitimate and empowering way for creators to bring their dreams to life and secure funding for their projects. There are entire businesses and communities that use crowdfunding exclusively as their source of revenue for operations, and businesses that use crowdfunding to supplement their revenue and help organize product launches. While this can be very attractive for those who may be starting out and wanting to take advantage of a platform's reach, there are several things to consider before diving head first into a campaign. The first is which platform is right for your project? There are several out there, so let's examine a few of them. # Kickstarter The "grand-daddy" of crowdfunding platforms, Kickstarter is largely to thank for the concept's normalization in the art and manufacturing sphere. Kickstarter offers some significant advantages: * Funds are only withdrawn if a project is fully funded * Easy tracking of backers through surveys * An established and reputable history of projects There are of course drawbacks, Kickstarter itself will take a portion of funds raised (this ranges from 5-10% depending on where you are located), and the funds withdrawn are pooled to include shipping if you are funding a physical release. So, if you were selling a game for \$40 and shipping cost \$15, the percentage collected would be from the full \$65 for each game. This is also factored for the funding goal, so if you were hoping to sell 100 units of the aforementioned game, your end goal wouldn't be \$4000, since that wouldn't factor in the shipping or the percentage of Kickstarter's take. More on that topic in coming articles. Finally, the funding not being released unless the goal is met is a double edged sword. While it makes sure you hit your minimum product goals for profitability, a fair amount of effort might be needed afterwards for collecting backer information since it's not gathered unless a campaign is fully funded (this happened with my game, where I had to hunt down 1 in 5 backers for addresses). There is also the problem with backers requesting refunds before a campaign has closed or sometimes their credit card information being declined. # Indiegogo Indiegogo is a competitor to Kickstarter and offers a couple of different approaches to the funding schemes. Firstly, backers cannot refund their pledges, so some of those inherent challenges in Kickstarter are not apparent. Indiegogo also offers the option to collect funds regardless of whether you reach your funding goal or not, which can be useful for games focused solely on Digital Distribution. From the experiences I've gathered from other crowdfunding users, this platform

seems better suited for productions or projects that can be completed regardless of their funding (think films or books) in that the creators would have made the product anyways, this just provides them some better means. This may be better suited for something like hiring artists or musicians for your game. # Ko-fi While I haven't seen any Game Boy projects funded via Ko-fi, I have seen and backed several other projects that have opted for this platform. Ko-fi is really simple, it's essentially a place to collect donations via PayPal, under the idea that you're buying someone a "coffee" in supporting them. While you could simply setup a "paypal.me" link and ask people to contribute, it's nice to be able to direct people to a place that includes information about the project, a way to update contributors (there is a blog/update feature) and you can set goals that people can track how much has been funded. The advantage to Ko-fi is it's 100% free – they don't take any cut of donations (although PayPal does itself), they simply expect users of the site to donate to them if they chose. The major disadvantage is there is no built in tracking. People can also donate anonymously meaning you have to do all the legwork yourself in terms of surveying people and collecting other data. # Pre-Order While this isn't technically a crowdfunding platform, it is a model that has worked for several creators and creates a workflow that you have the most control of. There are many options for creating websites and marketplaces that accept and process payments, and by setting up a pre-order period you can have an accurate idea of how much product you will need to have manufactured, plus you still get the funds up front. The best of these services (Shopify comes to mind) do charge both a monthly fee and a transaction fee, plus hosting a site has its own costs and challenges, so this is probably best suited for developers who already have that infrastructure in place. # Is It Worth It? In the end, securing funding is only a small portion of all the work that will go into a successful production run so think hard about all the aspects of making a physical game that will need to be implemented before taking on the project. There is the designing of materials, the production of carts and boxes/manuals, creating shipping labels and packing the product, on top of all the marketing and campaigning. All this work may not also yield high profits, so your time may not be well rewarded financially. We'll dive deeper into all these areas in this series, with the next article focusing on Production Costs and Planning. *Cover Art by Cinnristreusel*
Emi Paternostro

Emi Paternostro

Audio Engineer, Mac Technologist and Video Game Developer. Managing Editor of GBStudio Central. (he/him)

proximitysound.com ### Share this:

- [Click to share on Bluesky \(Opens in new window\) Bluesky](#)
- [Click to share on X \(Opens in new window\) X](#)
- [Click to share on Facebook \(Opens in new window\) Facebook](#)
- [Click to share on Mastodon \(Opens in new window\) Mastodon](#)
- [Click to share on LinkedIn \(Opens in new window\) LinkedIn](#)

- [Click to share on Reddit \(Opens in new window\)](#) [Reddit](#) [### Related](#)
-

Using Itch To Host Your Games

If you're looking to put your GB Studio game on the web for public (or private) consumption, Itch.io is a great resource. Not only do they have a wonderful community full of game jams and inspiration, but they will host your project for free. In this article, we'll show you the process from start to finish to get your build online. Before we begin, you're going to want to make sure your game is set up correctly. You may have been debugging your title to determine whether or not your game works perfectly, but you need to ensure the build is playable for others, so get that set up before starting the export process. Make sure your player character is on the correct starting scene and that all your variables are in order so the players get a great first experience. # Exporting from GB Studio – Web Build or ROM Getting a web build or ROM exported is fairly simple within GB Studio. All you have to do is click the “Export As...” button at the top right of the main screen (looks like an arrow pointing out of a box).. Within that menu, you'll need to click “Export Web” if you want to make the game into a web build that someone can play on their browser – GB Studio will make a folder inside your project's *build* folder called “web.” It will also open the folder it just made to show you the files. To prepare the files for Itch.io, you need to zip all of the files in this web folder into a .zip file, and you can name it whatever you wish (on both macOS and Windows, creating a zip file should be as easy as right-clicking and selecting compress, but there are also apps and tools that can do this). If you want to only export a ROM, you can click “Export ROM” instead, and GB Studio will take a moment to compile it, then output it to a “rom” folder inside of the *build* folder in your project. Again, it will open it to show you. The ROM will be called game.gb, and you can rename it now or can rename it later when you upload it to itch.io. # Uploading to itch.io Once you have your .zip file or ROM ready to upload, you'll need to go to the dashboard of your Itch account. If you don't have an Itch account, you can create one here. On your creator dashboard, click “Create New Project.” Here you can fill out the information about your game such as the title, description, add screenshots, set prices, all important considerations. Before you upload your game, you need to tell Itch what type of game you're uploading. If you want to upload a web build, find the section that says “Kind of project” and change it to HTML. If you are only going to offer a ROM to download, leave it as “Downloadable.” Next, turn your attention to the section that says “Uploads.” Click on the button that says “Upload files” , then navigate to and select your .zip file or ROM. If you uploaded a ROM, congratulations, your file is ready to go! You can choose to change your display name for the game, set a price, fill out the rest of the information and then move on to upload the web build. If you only want to have it available as a downloadable game, then skip ahead to read about getting paid. If you uploaded a web build, check “This file will be played in the browser”

under the file options and continue reading to learn more about the process. #

Web Build Settings There are a couple more options for the web build process that you should know about. First off, the game runs in a viewport on your project's page, and you can choose what resolution you want to display it at. If you go with the default resolution (640×360), you will have black bars on both sides of your game. In order to maximize visibility for your players, you can choose a resolution such as 480 (width) x 432 (height) to make the game run in a square format. You could also double the resolution to fill the horizontal space on the page (960×864), but it's almost obnoxious at that size. Did you know that your web build can run like a mobile emulator on someone's smartphone? It actually works pretty well. Set the embed option "Mobile friendly" to turn on this feature. You can choose in the embed options to automatically start the game on page load, which is simply a personal preference, and add a fullscreen button to your game that shows up in the bottom right corner so players can maximize the game on a desktop device. #

Getting Paid Itch has two ways for you to collect payments for your hard work. People that download your game can donate whatever they choose with a suggested donation, or they can just pay a minimum amount for the download. Keep in mind that a web build can be played freely and other than setting up your own solution, you cannot charge players to play your game online. Once someone buys or donates to you for downloading your game, you can see the information on the Payments page on your Dashboard. There are two options: paid later or direct to you. Itch.io defaults to paid later and will use their own payment provider to pay your final totals, less fees. You can access and change this option in your account settings, found when you click your profile drop-down menu on the top-right of Itch's homepage and select "Settings" under the "Account" subheading. Keep in mind that there is a required Tax Identity check that is a flat fee of \$3.00 for this option, and you will also have to wait 7 days to claim payments after they have been completed. You can access these options from Itch's Payouts page on the dashboard. *Note from the Editor:* If you reside outside of the US, there is a portion of the Tax interview where it asks for a Tax Identification Number to get a tax exemption. Use your country's equivalent of a Social Security Number (for example, a Social Insurance Number in Canada) in this place to avoid having 30% of your sales collected for US income taxes. If you choose to get paid "direct to you", you need to set up a payment processor, either Paypal or Stripe, and provide the necessary information to set up for transfers. There's a Payment modes guide on Itch's site to provide you more information about setting that up and what to expect. Make sure you know what you want to use and have it set up *before* uploading and listing your game for sale, or you will likely have to go through the additional processes to have Itch pay you directly. #

Final Thoughts After uploading your game, be sure to change your game's description to give players important information about your game (controls are a great thing to add here), and load the page up with screenshots, which can include gifs. With an uploaded build of your game, you can now easily share it with others by providing them a link to your page. Itch.io is such a great resource for allowing people to test out your game – even when it's not

complete, and is a valuable tool to find like-minded developers and other games to derive inspiration from. Even better, it's a great way to get paid for your work without paying extravagant developer fees! Good luck! Pearacidic

Pearacidic

Writer, Game Designer, Engine Bender, Retro Enthusiast. Co-creator of the In The Dark series and Grimace's Birthday. (He/Him)

pearacidicgames.com/ ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### *Related*

Version Control and GB Studio: Getting Started

Have you ever used undo enough times to hit the program limit? Maybe you are saving multiple copies of your project as a way to create backups? Without a doubt, making games can get messy. There's nothing worse than breaking a project, and having no way to go back to a working version of it. Thankfully, every GB Studio developer can use a tool to help them manage things: **Version Control**. Version control lets you go back in time to a previous version of your project. If some of your files go missing, your code breaks, or your files get corrupted, you can restore it to a previously functioning state. It's a lot like using save states in a Game Boy emulator. If you're afraid of losing a lot of progress by dying, you can make a save state to ensure you can go back and try again. In Git, a popular version control system, these save states are called **commits**. There are many services that utilize Git's system for tracking changes, but for this guide we'll be using GitHub, one of the most popular websites for version control. This guide will show you how to backup your project, and make a **commit** when you want to update your project's state to the site. You will need a Github account (which is free for most of its features) and a GB Studio project. If you'd like to follow along with something risk-free, try using one of the Sample Projects. To create a repository for your GB Studio project, in Github: 1. Click the + at the top right of the screen 2. Click "New repository" 3. Name your new repository 4. Choose Public or Private 5. Click "Create repository" **Note:** *If your project has personal information, or assets you don't want to make public, make sure to set your repository to private.* You should see a new page for your empty repository. Github offers a lot of ways to upload your files, but right now we'll simply use the website. 6. Click "uploading an existing file" in the Quick setup box 7. Drag your entire project folder into the space provided 8. Type in a message for your commit 9. Click

“commit” Commit messages are notes that help describe what you added to the project. This will make it easier to tell what’s changed if you have to go back and find a previous snapshot of your repository. “Adds player shooting script” or “adds town background” are some examples. Look here for how to write *really* good commit messages. You have now backed up your GB Studio project to Github. If you want to update this project, click “Upload files” on your repository page, then repeat steps 7-9. **Note:** *I like to give my project files absurdly short names like “ttm_a1”, yours should show up as whatever you named your project in GB Studio.* Using the Github website to update your files is not a great long-term strategy. Version control is best when you can make it a part of your project workflow. An example workflow could be: 1. Work on your GB Studio project 2. Open a program like Github Desktop 3. Make a commit 4. Click “push” This backs up your whole project privately to an external location, making it safe in case of hardware failure or broken code. Congratulations! You now know how to use Github to backup your projects and create commits to demark versions. Next time, I’ll show you how to restore previous contents of your repository, and introduce you to using clients for git. This will make it much easier to rollback, move, and rename your project files if you need to.

Ember

Musician and GB Studio Composer. Manager of the GB Studio Community Assets Collection. Contributor to the GB Studio Documentation project. (she/her & they/them)

deertears.itch.io/ ### Share this:

- Click to share on Bluesky (Opens in new window) Bluesky
- Click to share on X (Opens in new window) X
- Click to share on Facebook (Opens in new window) Facebook
- Click to share on Mastodon (Opens in new window) Mastodon
- Click to share on LinkedIn (Opens in new window) LinkedIn
- Click to share on Reddit (Opens in new window) Reddit ### Related

Conclusion

This comprehensive guide has traversed the diverse landscape of GB Studio development, equipping you with the knowledge and tools to create compelling Game Boy games. We’ve explored foundational concepts, delved into intricate design principles, mastered advanced scripting techniques, and understood the nuances of music, art, and hardware compatibility.

Remember that game development, especially within the charming constraints of the Game Boy, is a journey of continuous learning and creative problem-solving. Embrace experimentation, leverage the vibrant GB Studio community, and don’t be afraid to push the boundaries of what’s possible. Your unique vision,

combined with the power of GB Studio, holds the potential to craft the next nostalgic classic.

Happy developing!

Glossary

Actor: An interactive element in GB Studio, such as characters or objects. **Animation States:** Different visual representations or actions of a sprite. **Chiptune:** Music created using the sound chips of vintage computers or video game consoles. **Color Mode:** A GB Studio feature that allows games to display colors, unlocking more visual possibilities. **Custom Event:** Reusable blocks of events that can be called from different parts of a game. **DMG:** The original Game Boy model (Dot Matrix Game). **Duty Channels:** Two channels in the Game Boy sound chip primarily used for melodies and leads. **Envelope:** Controls how an instrument's volume changes over time. **Flashcart:** A cartridge used to play ROMs on original Game Boy hardware. **GB Studio Tracker:** A music editor within GB Studio for composing game music. **GBC:** Game Boy Color. **GBVM (Game Boy Virtual Machine):** The core scripting language of GB Studio 3.0+, allowing for advanced game logic. **hUGETracker:** A standalone music tracker software native to the hUGEDriver, used for creating Game Boy music. **Hexadecimal (Hex Values):** A base-16 number system often used in low-level programming and for defining effects in music trackers. **Metasprite:** A technique to combine multiple smaller sprites to form a larger, more complex sprite. **Noise Channel:** A channel in the Game Boy sound chip used for percussive sounds and effects like static, explosions, or wind. **Palette:** A set of four colors used to render graphics in GB Studio. **Pattern (hUGETracker):** A sequence of 64 notes/commands for a single channel in hUGETracker. **Portamento:** A smooth gliding from one pitch to another. **ROM:** Read-Only Memory, typically referring to game files or cartridges. **SGB Mode:** Super Game Boy mode, allowing Game Boy games to be played with color and custom borders on a Super Nintendo. **Sprite Editor:** A tool in GB Studio for creating and managing complex sprite animations. **Sprite Tiles:** The individual 8x8 pixel blocks that make up sprites. **Subpattern:** A sequence of commands fired off when a note is played in hUGETracker, impacting it in real time. **Tile:** An 8x8 pixel graphic unit used to construct backgrounds and sprites. **Tile Swapping:** Dynamically changing background tiles during gameplay. **Tracker:** A music editor where music is arranged in a grid-like interface. **VGM (Video Game Music):** A file format for storing video game music data. **Wave Channel:** A versatile channel in the Game Boy sound chip capable of producing a wide variety of sounds, including bass notes, kick drums, and recorded samples.

Quick Reference Cheat Sheet

GB Studio Editor Shortcuts

- **Ctrl+8** (or **View > Settings**): Open Settings (used for Color Mode options).
- **Ctrl+2**: Navigate directly to Sprite options.
- **Ctrl+6**: Access the Palette Editor.
- **Z**: Activate the Colorizer Tool (when in a scene).
- **Shift + Click**: Select multiple sprites in the Canvas (Sprite Editor).
- **Shift + Click** (in Colorizer Tool): Draw a straight line between two points.

hUGETracker / GB Studio Tracker Shortcuts

- **Keyboard (Q, A, Z, etc.)**: Play/place notes in the tracker.
- **Del**: Removes the note under the cursor.
- **Backspace**: Erase the note under the cursor and shifts notes under it up one.
- **Ctrl-C / Cmd-C**: Copy selected items (e.g., notes, frames).
- **Ctrl-V / Cmd-V**: Paste copied items.
- **Space Bar**: Switch between read and write mode in the main tracker area.

General Tips

- **Comments (GBVM Script)**: Use a semicolon (;) to denote comments.
- **Line Break (GBVM Script)**: Each line of code is ended with a line break. Accidentally hitting return/enter on your keyboard mid command and adding a line break will cause an error when it comes to compiling.
- **Sprite Optimization**: If you want to have more complex backgrounds or more sprites on screen, optimize sprite tile usage by composing complex sprites from smaller components.
- **Tile Priority (Color Mode)**: Create a layering effect where actors appear partially hidden behind the background, except for the first color in your palette (which remains visible). Only works in Color Mode.
- **Hide Layers Button (Colorizer Tool)**: Hides all overlays, triggers, links, and actors, so you can paint your scene without distractions.

Index

- "Hello World" Four Ways
- A Guide to SGB Mode
- Advanced Trigger Techniques
- Animating Background Tiles (3.0.3)
- Animating Background Tiles (3.1)
- Basics: Backgrounds

- Basics: Creating a HUD
- Basics: Custom Events
- Basics: Enemy Movement
- Basics: Flags
- Basics: Math Expressions
- Basics: Math Functions
- Basics: Recording with OBS
- Basics: Switches
- Basics: The Sprite Editor
- Basics: Trigger/Positional Menus
- Basics: Variables
- Breaking Barriers: Develop GB Games For Gamers With Visual Impairments
- Breaking Barriers: How To Develop Accessible GB Games For Gamers With Disabilities
- Compiling GB Studio Roms into Standalone Executables
- Converting Images for GB Studio
- Converting your GB Studio ROMs for use with the N64
- Create A(n almost) Seamless Transition
- Creating A "Reward" Scene
- Creating "Name Labels" for Dialogue Boxes
- Creating Cheat Codes
- Creating Multiple Dialog Frames using UI Palette Tricks
- Dealing With Burnout
- Development Workflow - Chapter 1: Researching Your Game
- Development Workflow - Chapter 2: Brainstorming Ideas
- Development Workflow - Chapter 3: Prototyping your Game
- Development Workflow - Chapter 5: Discipline & Demotivation
- Development Workflow - Chapter 6: Readability (Part 1)
- Development Workflow - Chapter 6: Readability (Part 2)
- Development Workflow - Chapter 7: Color Theory
- Development Workflow Chapter 10: Working As A Team
- Development Workflow Chapter 4: Level Design (Part 2)
- Development Workflow Chapter 8: Replayability
- Development Workflow – Chapter 9: Creating a Color Palette (Part 1)
- Development Workflow – Chapter 9: Creating a Color Palette (Part 2)
- Dissecting Classic Cutscenes From the Game Boy
- Distinguishing Reflashable Bootlegs
- Dynamic Auto Scroller
- FAQs for GB Studio Music
- GB Game Cartridge Shells
- GB Studio Music Lesson 1: Terminology and Basics of the Game Boy Sound Chip
- GB Studio Music Lesson 2: Noise Channel / Drums
- GB Studio Music Lesson 3: Duty Channels
- GB Studio Music Lesson 4: Wave Channel

- GB Studio Music Lesson 5: Effects
- GB Studio Music Lesson 6: Advanced Drums
- GB Studio Music Lesson 7: Special Tricks Part 1
- GB Studio Music Lesson 7: Special Tricks Part 2
- GB Studio Music Lesson 8: Common Mistakes
- GB Studio: Getting Started with Games – Chapter 1
- GB Studio: Getting Started with Games – Chapter 2 "Colorizing"
- Game Boy Camera+ Mod
- Game Boy Color Modes
- Getting GB Studio Games Running on the Xbox 360
- Getting Started in hUGETracker
- Getting Started with Games – Chapter 3: Cannon Defense
- Getting Your GB Studio Games Running on the Wii U
- Getting our GB Studio games running on the Nintendo DSi
- Getting our GB Studio games running on the Nintendo DS
- Getting your GB Studio Games on Real Hardware
- How I got Game Boy Games Running on the PS3
- How to Debug Your Game
- How to Make a Responsive Action Platformer
- How to Play a Game Boy Game on a Game Boy
- How to Use QR codes to Enhance Your GB(C) Games
- How to make a Game Boy SHMUP with GB Studio without losing your mind
- How to: Multi-Language Mechanic
- HugeTracker Part 6: Subpatterns
- hUGETracker Effects Part 2: Tempo, Pattern Break and Pattern Jump and Stereo Panning
- hUGETracker Lesson 2 : Noise Channel
- hUGETracker Lesson 3: Duty Channels
- hUGETracker Lesson 4: Wave Channel
- hUGETracker Lesson 5: Effects - Portamentos, Volume Control and Note Cuts
- Incrementing and Decrementing Values Like a Pro - How to Use The Min/Max Functions to Save Time
- Learning GBVM
- Let's Build an RPG | Quest 1-1: Battle System Player Turn
- Let's Build an RPG!
- Let's Build an RPG: Quest 2 - Rewards & Level Up
- Let's Program a Puzzle Game! - Part 1
- Let's Program a Puzzle Game! - Part 2
- Let's Program a Puzzle Game! - Part 3
- Let' s Build an RPG | Quest 1-2: Enemy Turn, Effects, and Stats
- Let' s Build an RPG | Quest 1-3: Animations & HUD
- Lights Out - DMG Custom Color Palettes Tutorial
- Managing Sprite Tiles
- Making a Cutscene for the Game Boy in a Week

- Making an Isometric Game in GB Studio
- Making a Theremin Sound in GB Studio
- Modifying the GB Studio Engine
- Playing GB Studio Games on a 3DS
- Playing GB Studio ROMs in Minecraft
- Porting GB Studio Games to Dreamcast
- Porting GB Studio Games to the GBA with GBSenpai
- Possibly the WORST way to play Gameboy Games
- Quest 3: Back to Town Part 1 - The Inn
- Reflashing GBA Bootlegs with an NDS or NDS Lite
- Running My GB Studio Games on Xbox
- Screen Sized Bosses
- Slopes in GB Studio
- So You Want To Crowdfund? Part 1
- So You Want To Crowdfund? Part 2
- So You Want To Crowdfund? Part 3
- So You Want To Crowdfund? Part 4
- Sound Effects and Music in GB Studio 3.1
- Sprite Style Differences in Game Boy and Other Consoles
- Ten Lessons I learned During My Latest Project with GB Studio
- Testing Your Game on Different Hardware
- Text Input Systems
- The Goomba Color Emulator
- The Problem with Random
- Tile Swapping 2.0
- Top Five Mistakes for First Time GB Studio Users
- Translate your games with GlyphBoy Swap
- Understanding Fonts
- Understanding GBVM
- Unraveling The Thread Events
- Upgrading the GBS Platformer Engine
- Using Itch To Host Your Games
- Variables and Maxing Them Out
- Version Control and GB Studio: Getting Started
- Version Control and GB Studio: Reverting Changes
- Version Control and GB Studio: Using GitHub Desktop
- VGM Sound Effects Part 2
- VGM Sound Effects Part 3
- VGM Sound Effects Part I
- When Random Is NOT Random
- Zen and the Art of Retro Level Design in “Kudzu”
- Your First GB Studio Game